TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: M 2612 - Elektrotechnika a informatika

Studijní obor: 3902T005 - Automatické řízení a inženýrská informatika

Síťové grafy a hierarchické struktury v Oracle

Networks and hierarchical structures in Oracle

Diplomová práce

Autor: **Martin Horák**

Vedoucí práce: RNDr. Klára Císařová, Ph.D.

Konzultant: Ing. Karel Šredl

V Liberci 16. 5. 2008

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Abstrakt

Diplomová práce je zaměřena na problém zpracování síťových grafů a stromů v prostředí relačních databází, které jsou známé tím, že právě pro hierarchicky strukturovaná data nejsou ideální. Na základě několika článků na internetu o stromech a téměř žádných informací (kromě interních informací z firmy) o zpracování síťových grafů bych měl vytvořit ucelený text, který se bude opírat o mé vlastní výsledky a ve kterém budou popsány hlavní přístupy k práci s hierarchickými strukturami v relačních databázích, zejména pak s důrazem na implementaci v databázovém systému Oracle.

Část týkající se síťových grafů je přímo firemním zadáním, které očekává efektivní zpracování časové analýzy stavebních projektů. Navíc, vzhledem k nedostatku opravdu kvalitních a ucelených informací (například v odborných publikacích) o stromech a síťových grafech v databázích, by měla být rozšířena firemní báze znalostí.

Abstract

The thesis is focused on a problem of handling networks and trees in the environment of relational databases, which are known for not being ideal for storing hierarchically structured data. On the basis of a few articles on the internet about trees and almost no information, but the company´s, about networks, I should create a complex text based on my own results which would describe possibilities of implementation and processing of hierarchical structures in relational databases, pinpointing some special features of Oracle database system.

The part regarding networks was ordered by a company that expects an effective processing of time analysis of construction projects. Moreover, as there is almost no comprehensive information both about networks and trees in databases, the company´s internal knowledgebase should be extended.

Obsah

1 Úvod

Teorie grafů je poměrně mladý matematický obor, který hraje zásadní roli v dnešní informatice. Kdo dnes nepoužívá elektronické jízdní řády, mapy, navigační systémy nebo dokonce samotné počítače? Ve všech zmíněných zařízeních se vyskytuje člověku užitečná aplikace teorie grafů. Začněme ale popořádku…

První zmínka o teorii grafů sahá až do 18. století, v roce 1736 vyřešil švýcarský matematik Leonard Euler populární problém „Sedmi mostů města Kaliningradu“ a nejen to, Euler navrhl řešení všech podobných úloh, tedy lze-li graf nakreslit „jedním tahem“.

Další pokroky (tehdy ještě nevědomé) v teorii grafů přišly až v 19. století. Vědci Arthur Cayley a Gustav Kirchhoff použili grafy při studiu svých vědních oborů a tyto aplikace se používají i dnes. Cayley při studiu chemie použil grafy ke znázornění atomů (kroužky) a vazeb (hrany), Kirchhoff řešil problém toků v sítích. Avšak nejznámější problém předložil Francis Guthrie v roce 1852. Jde o „Problém 4 barev“, tedy jestli lze pouhými čtyřmi barvami vybarvit politickou mapu světa. Zajímavé je, že tento problém byl vyřešen až o více než sto let později počítači.

A právě až s věkem počítačů přišel obrovský rozvoj teorie grafů, uznání jako části diskrétní matematiky a sjednocení názvosloví. Dnes je tato teorie vůbec nejrozšířenější a nejvyužívanější částí diskrétní matematiky.

Proto není s podivem, že většina moderních programovacích jazyků má buď nativní podporu pro grafové struktury (tedy hlavně stromy), popřípadě jsou známé algoritmy k jejich implementaci a používání. U databázových systémů je ale situace odlišná – do nedávna ani ty nejvýkonnější přímo nepodporovaly grafové struktury a navíc algoritmy existující pro procedurální jazyky nejsou pro databázové systémy vhodné nebo se nedají použít vůbec.

Přesto ale velice často potřebujeme graf, strom nebo síťový graf do databáze uložit a pracovat s ním a navíc samozřejmě co nejefektivněji. Proto vzniká tato diplomová práce, jejíž autor si klade za cíl zpracovat možnosti implementace a algoritmů pro zacházení se stromy a síťovými grafy (tedy nejběžnějšími strukturami) v databázích.

Diplomová práce byla zadána firmou StringData, s.r.o., v rámci projektu Contec. Projekt je zaměřený na stavební plánování, při kterém se využívá síťových grafů. Jedním z cílů diplomové práce je tedy zpracování co nejefektivnější metody pro časovou analýzu síťových grafů, jež se stane součástí zmíněného projektu.

2 Stručný úvod do teorie grafů

Teorie grafů je komplexní teorie, jejíž zpracování mnohonásobně přesahuje rámec diplomové práce. Já se ve svém stručném úvodu budu soustředit pouze na ty části, jež pomohou definovat stromy a síťové grafy, popřípadě další pojmy, které budou využity v praktické části. Jistě je ale důležité část teorie obsáhnout a umožnit tak čtenářům orientovat se v příslušné terminologii.

2.1 Definice grafu

2.1.1 Orientovaný graf

Orientovaný graf, který budeme značit G, je uspořádaná trojice $G = (V, E, \varepsilon)$, kde V je konečná neprázdná množina, jejíž prvky nazýváme vrcholy, E je konečná množina, jejíž prvky jsou orientované hrany a ε je zobrazení $\varepsilon : E \rightarrow V^2$. Zobrazení ε, které nazýváme incidentním zobrazením nebo prostě incidencí, přiřazuje každé hraně z E počáteční a koncový vrchol z množiny V. Říkáme, že vrcholy jsou incidentní s hranou e (a naopak).

2.1.2 Neorientovaný graf

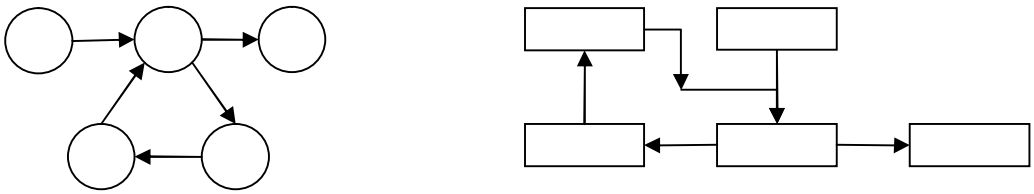
Někdy nezáleží na orientaci hran, nepotřebujeme rozlišovat počáteční a koncový vrchol. Neorientovaný graf bychom definovali stejně jako graf orientovaný s tím, že množina hran E obsahuje neorientované hrany. Stejně tak si lze neorientovaný graf představit jako orientovaný tak, že každé dva vrcholy jsou spojeny dvěma hranami s opačnou orientací. Toho se využívá například v situaci, kde u některých hran na orientaci záleží a u jiných ne.

2.1.3 Smyčka

Mějme počáteční a koncový vrchol $x, y \in V$ tak, že $x = y$, pak hranu incidentní s těmito vrcholy nazýváme smyčkou. Stručně řečeno, hrana spojuje vrchol se sebou samým.

2.1.4 Kreslení grafu

Mnohem dříve, než se o grafech začalo mluvit, se grafy kreslily. Vrcholy většinou zakreslujeme jako kroužky nebo obdélníky, popřípadě i jinak (např. součástky elektrických obvodů). Hrany se kreslí jako čáry spojující příslušné vrcholy. Z toho samozřejmě vyplývá, že každý graf lze nakreslit zcela libovolně a jeden graf může mít nekonečně mnoho znázornění.



Obrázek 1 - různá znázornění téhož grafu

Rovinný, nebo také planární graf je takový, který lze v rovině nakreslit, aniž by se libovolné dvě hrany překřížily.

2.1.5 Ohodnocený graf

Ohodnocený graf je speciální případ grafu, ve kterém hrany nesou nějakou hodnotu, například délku cesty, náklady na cestu, apod. Stejně tak mohou být ohodnoceny i vrcholy.

2.2 Sled, tah, cesta v grafu

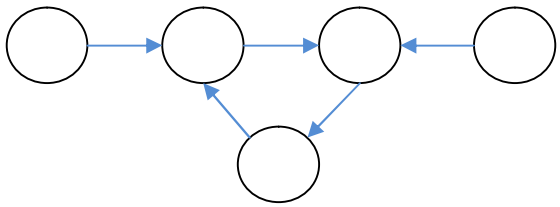
2.2.1 Orientovaný sled

Termínem orientovaný sled označujeme posloupnost vrcholů a hran $S = (v_0, e_1, v_1, e_2, v_2, \ldots, ek, v_k)$ takovou, že pro každou hranu platí $e_i = (v_{i-1}, v_i)$. Orientovaný je tento sled proto, že se zachovává jeden směr - „vpřed“.

2.2.2 Neorientovaný sled

Pro neorientovaný sled platí, že každá strana e_i spojuje dva sousední vrcholy, tedy $e_i = \{v_{i-1}, v_i\}$. Je to tedy zobecnění orientovaného sledu – každý orientovaný sled je zároveň i sledem neorientovaným.

Z výše uvedených definic vyplývají dva závěry. Ve sledech se mohou opakovat vrcholy i hrany. Navíc pojem „neorientovaný sled“ má smysl i v orientovaném grafu. Jak lze vidět na obrázku, kde posloupnost $S = (v_1, e_1, v_2, e_2, v_3, e_3, v_4, e_4, v_2, e_2, v_3, e_5, v_5)$ je neorientovaným sledem v orientovaném grafu. Pokud by hrana e_5 byla opačně orientovaná, šlo by o orientovaný sled.



Obrázek 2 - Neorientovaný sled v orientovaném grafu

2.2.3 Tah

Sled (tedy orientovaný i neorientovaný) nazýváme tahem, jestliže se v něm neopakuje žádná hrana. Vrcholy se opakovat smějí.

2.2.4 Cesta

Cesta je tah, ve kterém se neopakuje žádný vrchol, nebo chcete-li, je to sled, v němž se neopakují ani vrcholy ani hrany. (Termín odpovídá jeho mluvnickému používání – na cestě z Liberce do Ústí nad Labem určitě nepojedeme dvakrát skrze stejné město.) I cesty rozlišujeme na orientované a neorientované.

Pojem cesty, zvláště pak její aplikace v ohodnocených grafech je jedním z vůbec nejdůležitějších pojmů v teorii grafů a řešení úloh nejkratších cest je jednou z nejřešenějších úloh diskrétní matematiky vůbec. Podotýkám, že v těchto úlohách jsou často užívány termíny „délka cesty“ a „dostupnost“. Délkou cesty rozumíme součet ohodnocení všech hran, které jsou součástí posloupnosti cesty. Dostupný vrchol *Cíl* z vrcholu *Start* je tehdy, existuje-li cesta, která oba vrcholy spojuje.

2.2.5 Cyklus

Specifický případ takzvané uzavřené orientované cesty, kde je počáteční vrchol *Start* a koncový *Cíl* stejný, nazýváme cyklem. Jedná-li se o neorientovanou cestu, mluvíme o kružnici.

2.3 Acyklický a cyklický graf

2.3.1 Acyklický graf

Acyklický graf je graf, který neobsahuje žádný cyklus ani smyčku.

Tato jednoduchá definice má několik zajímavých a užitečných důsledků:

1) V acyklickém grafu nelze nalézt sled, ve kterém by se opakovaly vrcholy nebo hrany – každý sled v acyklickém grafu je tedy cestou.
2) Existuje minimálně jeden uzel, do kterého nevede žádná hrana a minimálně jeden, z kterého žádná hrana nevychází.
3) Existuje takové topologické uspořádání vrcholů v_1, v_2, v_3 … v_k, pro které platí, že pro všechny hrany spojující vrcholy v_i a v_j platí $i < j$. Jednodušeji řečeno, můžeme vrcholy očíslovat, tak, že budou postupně uspořádané.

Hlavně třetí důsledek je nesmírně důležitý z programátorského hlediska, neboť nám říká, že acyklický graf lze uložit do pole (popřípadě do tabulky v databázi) „za sebou“. To samozřejmě vede k velkým úsporám při vyhledávání a procházení acyklických grafů.

Ale i ostatní dva důsledky jsou pro programátory velice užitečné. Jednak je vidět, že se nemusíme bát nekonečnosti cyklu při procházení grafu a jednak není třeba v acyklickém grafu kontrolovat sledy, jestli jsou cestami nebo ne.

2.4 Strom a síťový graf

2.4.1 Kořen stromu

Kořenem stromu rozumíme vrchol, ze kterého jsou orientovaně dostupné všechny ostatní vrcholy grafu. Zároveň se dá říct, že kořen stromu je vrchol, do kterého nevede žádná orientovaná cesta.

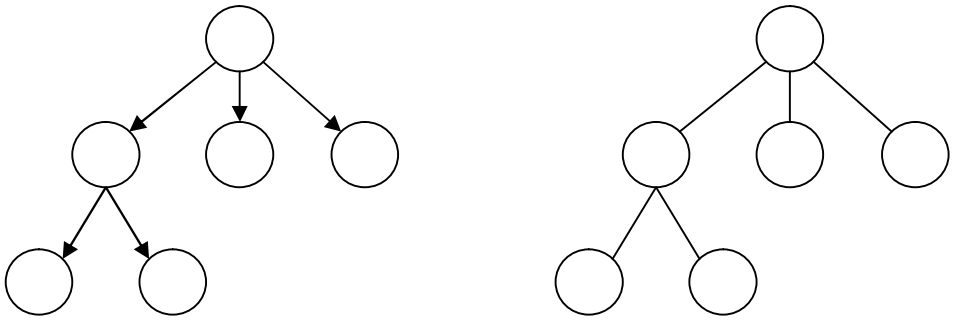
2.4.2 Kořenový strom

Kořenový strom je takový acyklický orientovaný graf, ve kterém existuje vrchol, kořen stromu r (z anglického „root“), který je spojen se všemi ostatními vrcholy křenového stromu právě jednou orientovanou hranou. Kořenový strom bývá označován také jako „větvení“.

2.4.3 Strom

Naproti tomu je strom acyklický graf (tedy neorientovaný), ve kterém jsou každé dva vrcholy spojeny právě jednou neorientovanou cestou. Tomuto jevu říkáme souvislost grafu.

I ve stromu často volíme vrchol r, i když v tomto případě nemusí být jednoznačně daný. Na obrázku je znázorněn rozdíl mezi kořenovým stromem a stromem, je vidět, že ve stromu si jako vrchol r můžeme označit libovolný vrchol.



Obrázek 3 - Kořenový strom (vlevo) a strom

2.4.4 Pohled programátora

Z programátorského hlediska je celá situace jednodušší, což vede ke značnému komfortu. Například se nerozlišuje strom a kořenový strom, implementace i práce s nimi je stejná. Pokud bychom šli ještě dál, můžeme prohlásit, že z programátorského hlediska je jedno, je-li graf orientovaný nebo ne – přeci i v orientovaném grafu se potřebujeme často pohybovat proti orientaci hran, takže ve výsledku s nimi pracujeme stejně.

I problém kořenu a jeho volby ve stromu je z hlediska programátora v naprosté většině případů jednoznačný, takže strom se stává kořenovým stromem s tím, že na orientaci hran nehledíme.

Vzhledem k zaměření této práce spíše na programátorské hledisko věci, budu i já používat jednotný termín *strom* pro strom i kořenový strom.

Další programátorskou specialitou je názvosloví týkající se stromů. Toto vychází pravděpodobně z jejich využití pro účely zakreslení rodokmenů, tzv. genealogických stromů. Například vede-li hrana z vrcholu v_i do vrcholu v_j, pak o vrcholu v_i hovoříme jako o rodiči (parent), nebo také otci. Vrchol v_j nazýváme potomkem (child), popřípadě synem vrcholu v_i. Vrcholům se stejným rodičem říkáme sourozenci (siblings). List je prvek, který nemá žádné potomky. I já tuto terminologii budu ve své práci dále používat.

Zvláštní případ stromu, který se v informatice hojně používá, je *binární strom*. To je strom, kde každý rodič má maximálně dva potomky. Těmto potomkům pak říkáme pravý a levý potomek. Jako zajímavost lze uvést, že binárních stromů používají překladače programovacích jazyků k reprezentaci algebraických výrazů.

Aby byla terminologie kompletní, nemohu nezmínit ještě pojem *výšky stromu*. Výškou stromu myslíme délku cesty (v tomto případě počet hran) vedoucích od kořenového vrcholu k nejzazšímu potomkovi. Tento potomek má pak největší *hloubku,* leží v poslední *vrstvě*. Pro názornost – na obrázku 3 mají oba stromy výšku 3.

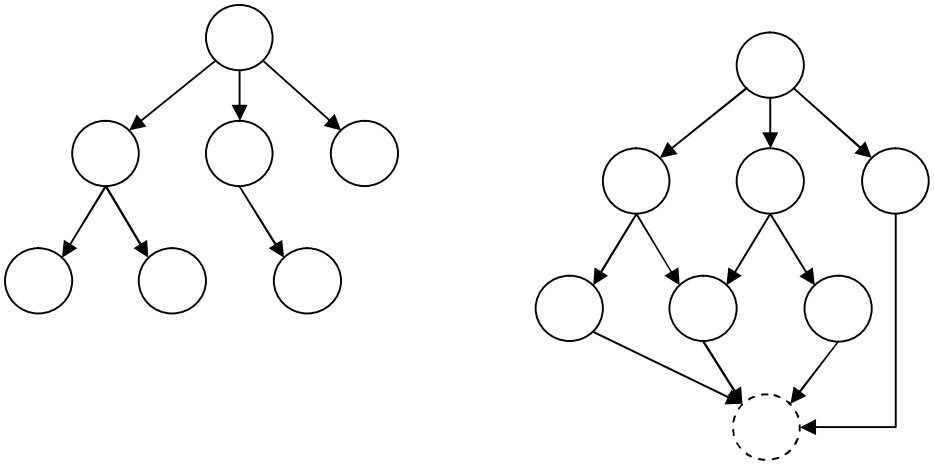
2.4.5 Síťový graf

Acyklický orientovaný graf s ohodnocenými hranami (popřípadě vrcholy) se nazývá síťový graf. Síťový graf nemusí mít kořen (ovšem zde mu budeme říkat *Počátek*), zpravidla jej však má, nebo se mu přidává i fiktivní. Stejně tak se síťovému grafu podle potřeby přidává i poslední vrchol *Konec*, podle potřeby též fiktivní.

Jak už názvy obou speciálních vrcholů napovídají, síťový graf slouží ke znázornění průběhu činností, nebo ještě lépe, k plánování složitějších činností, projektů. Jde o užitečnou reprezentaci navazujících činností, ohodnocení hran nebo vrcholů pak reprezentuje časovou nebo finanční náročnost. Zvláště pak v oboru stavebnictví je jeho užití nabíledni.

Se síťovými grafy souvisí řada pojmů, z nichž nejdůležitější je pojem *kritické cesty*. Je to nejdelší cesta v grafu (může jich být hned několik) a je to právě ta posloupnost činností, na které závisí výsledek celého projektu – ať už finanční nebo časový. Proto je na ni během projektování i realizace soustředěna maximální pozornost, mluvíme o řízení metodou kritické cesty (CPM – critical path method).

Ačkoliv to nemusí být ihned zřejmé, „nejpříbuznější“ k síťovým grafům jsou právě stromy. Nejzásadnějším rozdílem mezi oběma typy grafů je, že v síťovém grafu není zajištěno, že z počátku vede ke každému vrcholu právě jedna cesta – může jich vést hned několik. Další rozdíl je pak v existenci (respektive neexistenci) kořene-počátku v síťovém grafu. Skutečně, nezavedeme-li fiktivní počátek, síťový graf jich může mít hned několik! Poslední rozdíl je pak v existenci (i když někdy jen fiktivního) konce. Rozdíly jsou znázorněny na obrázku 5, jen podotýkám, že síťový graf je záměrně znázorněn shora dolů, namísto tradičnějšího zobrazení zleva doprava.



Obrázek 4 - Rozdíly mezi stromem (vlevo) a síťovým grafem

3 Databáze

Systém řízení báze dat (SŘBD) neboli databáze je rozsáhlá a hustě využívaná část oboru informatiky. Slouží ke strukturovanému ukládání téměř jakýchkoliv dat, proto ji najdeme skoro všude, kde se nějakým způsobem využívá informační technika. S její pomocí řešíme jednoduché úkoly, například ukládání příspěvků diskusních fór, e-shopy, ale i úlohy, které vyžadují obrovský stupeň zabezpečení a na nichž závisí nemalé finanční částky, například v bankovnictví.

Teorie databází je velmi propracovaná a obsahuje nemalé množství specifických termínů. Autor tohoto textu si neklade za cíl se jí zaobírat – jednak existuje velké množství kvalitních zdrojů a jednak by i nějaké smysluplné shrnutí vydalo na celou knihu. Budu tedy předpokládat jistou čtenářovu znalost tematiky, v případě nejasností či zájmu jej s dovolením odkážu na doporučenou literaturu.

Stejně tak se nemohu věnovat ani základům databázového jazyka SQL a jeho procedurálním rozšířením. Malou výjimku udělám jen v případě speciálních funkcí systému Oracle určených k práci se stromy, neboť dobrých zdrojů (v češtině) k tomuto tématu je pohříchu málo.

Jak už jsem nastínil v minulém odstavci, při psaní této práce jsem využíval hlavně prostředí databázového serveru Oracle. Nejenže je to požadavek zadavatele, ale zároveň je to v případě hierarchických struktur zatím nejlepší volba. Oracle zatím jako jediný (další databázové systémy příkladu Oracle následují) nabízí některé užitečné funkce pro práci se stromy. Ovšem i programátoři ostatních systémů nepřijdou zkrátka, budu se snažit na rozdíly vždy na příslušném místě upozornit.

4 Relační databázové systémy a hierarchické struktury

V historickém vývoji databázových systémů logicky vyhrál systém nejuniverzálnější, tedy relační databáze. A tak navzdory faktu, že jsme mohli vidět už několik lepších řešení ukládání hierarchicky členěných dat, musíme se dnes často potýkat s problémem uložení těchto dat právě v relačních databázích, pro tento typ dat nepříliš vhodných. O aktuálnosti problému vypovídá i to, že vývojáři Microsoftu chtěli již ve svém systému Windows Vista využít nový souborový systém založený na relační databázi.

Dříve se úkoly, nezpracovatelné klasickým SQL, musely řešit v jiných programovacích jazycích s tím, že programátor samozřejmě ještě řešil komunikaci mezi databází a použitým jazykem. Samozřejmě i dnes můžeme takto relativně snadno vyřešit jakoukoliv úlohu týkající se hierarchických struktur, ale naštěstí už existují i lepší a hlavně rychlejší alternativy.

Jednou z nich je bezesporu užití některého nadstavbového procedurálního jazyka, které nám některé DB systémy nabízejí. V případě Oracle je to jazyk PL/SQL, který je částečně kompilovaný (ve verzi 10g), což zaručuje poměrně vysokou rychlost. Další nespornou výhodou je, že nemusíme řešit přenos dat z DB do jiného jazyka, což je operace spojená s poměrně velkou časovou režií. Poslední důležitá výhoda plyne z typu jazyka, můžeme s přehledem používat tradiční a prověřené algoritmy pro práci se stromy.

Další možností je použití rekurzivního SQL dotazu. Samozřejmě jde asi o nejrychlejší řešení, přesto není zcela běžné. V databázových systémech založených na ANSI SQL nedostane programátor příliš prostoru – chybějí (zatím) nějaké užitečné funkce. V Oracle je situace o dost příznivější. Přesto narážíme na další překážku, a sice nepřehlednost takového dotazu. Proto je víceméně na každém programátorovi, čemu dá přednost. Je-li hlavním požadavkem rychlost, vyplatí se vytvoření nějakého rekurzivního dotazu. Pokud je naším cílem přehlednost a také přenositelnost kódu, je lepší sáhnout po PL/SQL.

5 Stromy

Stromy jsou jednoznačně nejvyužívanější hierarchickou strukturou, se kterou se setkal nebo se setká každý programátor. Naštěstí je implementace, uložení stromu do DB poměrně jednoduchou záležitostí, vždyť stačí jediná tabulka se sloupcem navíc, který se odkazuje na identifikátor svého předchůdce. Existují také různá rozšíření a vylepšení, která vždy ale mají svá úskalí. Výběr vhodné implementace tedy nelze doporučit obecně, ale vždy bude závislý na potřebách dané aplikace.

5.1 Sebereferenční tabulka

5.1.1 Implementace

Nejjednodušší způsob uložení stromu je jedna jediná tabulka, jejíž řádky reprezentují uzly stromu. V tabulce vytvoříme navíc sloupec `PARENT`, který bude reprezentovat vazby, zpětné na nadřazený uzel. Sloupec `PARENT` tedy bude odkazovat do téže tabulky, ve které se sám nachází, proto se jedná o sebereferenční tabulku.

```
create table tree
  (id        number not null primary key,
   name      nvarchar2(40),
   value     number,
   parent    number
  );

alter table tree add constraint FK_tree foreign key (parent) references tree (id);
```

V tomto případě je ještě vhodné vytvořit index nad sloupcem `PARENT`, neboť podle něj budeme často vyhledávat (a získáme cca 10-ti násobné zrychlení pro vyhledávání podle tohoto sloupce).

```
create index ix_tree_parent on tree(parent);
```

Tato základní implementace trpí několika neduhy, z nichž jde hlavně o problematický výpis a odebírání prvků, respektive celých větví stromu. Ani nalezení cesty od uzlu ke kořeni se neobejde bez nějaké rekurzivní funkce na aplikační úrovni. Na druhou stranu, jednoduchost a hlavně rozšiřitelnost jsou hlavní výhody této implementace, stejně jako možnost použít rekurzivní SQL (v Oracle tedy klauzule `CONNECT BY – START WITH`).

5.1.2 Rozšíření implementace

Přidáním atributů – dalších sloupců – do sebereferenční tabulky ji můžeme vhodně rozšířit a zjednodušit si tak naši práci, popřípadě ji markantně urychlit. Ovšem vždy s sebou

nese rozšíření i některá negativa, proto je nutné si vždy důkladně promyslet, jaké operace budeme provádět, budou-li se často měnit data, atp., a podle toho se pak rozhodnout, které rozšíření využít.

Přidání atributu následník (vytvoření vazební tabulky)

Chceme-li přidat do atributů uzlu stromu i jeho následníky, narazíme na problém – nevíme vlastně, kolik následníků uzel bude mít. Abychom se vyhnuli nekoncepčnímu řešení s několika sloupci pro následníky (i když třeba známe jejich maximální počet, nikde není řečeno, že se toto maximum nemůže změnit), sáhneme raději ke spojovací tabulce, která nám příslušné reference zajistí. V naší tabulce `TREE` už samozřejmě nebudeme potřebovat sloupec `PARENT`, neboť i tuto informaci ponese nová spojovací tabulka.

```
Alter table tree drop column parent;

create table tree_connections
  (id        number not null primary key,
   parent    number,
   child     number
  );



alter table tree_connections add constraint FK_tree_conn_parent foreign key
(parent) references tree (id);
alter table tree_connections add constraint FK_tree_conn_child foreign key (child)
references tree (id);



create index ix_tree_connections_parent on tree_connections(parent);
create index ix_tree_connections_child on tree_connections(child);
```

Takováto konstrukce nám může být opravdu v mnohém užitečná. Výpis větve stromu už můžeme realizovat nerekurzivní procedurou, úpravy celého stromu (nebo opět jeho větve) lze pak provádět jedním SQL dotazem bez rekurze, popřípadě opět přehlednou dopřednou procedurou. Navíc, máme možnost přiřazovat vazbám další argumenty pomocí dalších sloupců spojovací tabulky, což může být užitečné nejen v případě stromů, ale, jak uvidíte dále, i v případě obecnějších struktur – tímto způsobem lze totiž i realizovat uložení síťového grafu.

Vykoupením za dobré vlastnosti tohoto rozšíření je nutnost používat spojení (JOIN) tabulek a obtížnější odebírání prvků (vlastně jsme si situaci zkomplikovali i pro případ jednoho odebíraného uzlu).

Přidání atributu LEVEL

Přidáním atributu LEVEL (hloubka) si podstatně zjednodušíme hlavně výpis stromu, na který nám pak bude stačit jeden dotaz do databáze. Samozřejmě se také usnadní, respektive zrychlí hledání uzlů – následníků, když budeme vědět, v jaké hloubce hledat. Někdy je také potřeba vyhledávat uzly ve stejné hloubce, k čemuž je tento atribut přímo určen.

V Oracle je situace o to jednodušší, že při použití rekurzivních dotazů máme pseudosloupec LEVEL přímo k disposici, a nemusíme tak vymýšlet proceduru pro jeho vyplnění. Následující dotaz vytvoří pohled, který je poměrně slušným výpisem stromu.

```
create or replace view vw_tree
as
select
  lpad(' ', (level - 1)*2) || name as padded_name,
  id,
  parent,
  level as the_level  -- „level" je rezervované slovo, použijeme the_level
from tree
connect by prior id = parent
start with id = 1;

select * from vw_tree;
```

```
PADDED_NAME                      ID     PARENT  THE_LEVEL
------------------------------ ---------- ---------- ----------
Moje hudba                              1                     1
  Zpevaci                               2          1          2
    Jarek Nohavica                      3          2          3
      Rok Dabla                         4          3          4
      Mikymauzoleum                     5          3          4
      Osma barva duhy                   6          3          4
    Karel Kryl                          7          2          3
      Monology                          8          7          4
      To nejlepsi                       9          7          4
    Manu Chao                          10          2          3
      Proxima Esperanza                11         10          4
  Skupiny                              12          1          2
    Schodiste                          13         12          3
      Mokry Pradlo                     14         13          4
      Svinska Przola                   15         13          4
    Hm..                               16         12          3
      Ehm                              17         16          4
      To by mohlo byt zajimave         18         16          4
      Obed                             19         16          4

19 rows selected
```

Ke kódu (respektive k rekurzivnímu SQL) se vrátím až v kapitole „Práce se sebereferenčními tabulkami", prozatím jej tedy ponechám bez komentáře.

Přidání cesty (genealogický identifikátor)

Velice oblíbená technika pro rozšíření sebereferenční tabulky je přidání sloupce, ve kterém se uchovává informace o cestě ke kořenu. Tuto cestu někdy nazýváme *genealogickým stromem (nebo identifikátorem)*, název pochází z původního užití pro grafické vyjádření rodinných vazeb.

Víceméně jde o stejný princip známý ze všech operačních systémů – takzvaná cesta k souboru, je vlastně genealogický identifikátor uložení souboru na disku, oddělovačem uzlů je znak zpětného lomítka „\“. My samozřejmě můžeme použít určitá zjednodušení, například neměnnou délku názvu jednoho uzlu, čímž si značně zjednodušíme práci (dokonce nám nebude třeba oddělovače). Zatím tedy postačí přidat jeden sloupec typu `VARCHAR2` do tabulky `TREE`.

```
alter table tree add genealogical_id varchar2(4000);
```

Ještě nám zbývá genealogický identifikátor naplnit hodnotami. Znamená to, že musíme každému uzlu přiřadit jednoznačný identifikátor s tím, že uzel vždy nejprve zdědí identifikátor svého předchůdce (v operačních systémech cesta k adresáři, ve kterém je soubor uložen) a následně je rozšířen o jednoznačný identifikátor sebe sama (v operačních systémech je to název souboru). V našem zjednodušeném případě tedy vezmeme řetězec předchůdce a doplníme jej. Kořen musíme samozřejmě ošetřit na začátku procedury.

Celou proceduru lze realizovat dvěma způsoby. Univerzálním, ale nepříliš rychlým řešením je rekurzivní funkce, která se dá shrnout do několika kroků:

1) Přidat jednoznačný genealogický identifikátor kořenu
2) Zavolat rekurzivní funkci s parametrem ID_rodiče (v prvním volání je tímto rodičem kořen)
3) V cyklu pro každého potomka:
    a) Zvýšit počitadlo potomků pro aktuálního rodiče o 1
    b) Aktualizovat záznam genealogického identifikátoru potomka (například písmeno „A“ + počet potomků aktuálního rodiče)
    c) Volat rekurzivně funkci z kroku 2

Poznámka: Paměťová, ale hlavně časová náročnost rekurze prudce stoupají s hloubkou stromu. Pro nehluboké stromy (do 5. – 6. úrovně) ji můžeme bez obav použít.

Druhá možnost, dopředná procedura, je sice o poznání rychlejší než rekurze, ale je závislá na tom, aby byl strom uspořádán (viz. Teorie acyklických grafů v kapitole 2.3), čili aby každý potomek měl `ID` – primární klíč vyšší než jeho rodič. Pokud strom takto uspořádaný máme, máme vyhráno. V opačném případě je třeba strom ještě utřídit, což s sebou bohužel nese velké časové požadavky.

Samotná procedura se dá opět naprogramovat podle několika obecných kroků:

1) Přidat jednoznačný genealogický identifikátor kořenu
2) V cyklu procházet postupně podle `ID` všechny uzly a pro každý aktualizovat jeho genealogický identifikátor tak, že k identifikátoru rodiče přidáme identifikátor vlastní, který bude od začátku abecedy posunutý o X, přičemž X = počet sourozenců, kterým byl identifikátor již přidělen.

Poznámka: Pokud budeme věnovat trochu času operacím nad stromy tak, aby tyto operace neporušily jeho konzistenci – uspořádání, lze náročnou proceduru pro uspořádání stromu spustit pouze jednou (při vytváření stromu).

Nejjednodušší implementaci genealogického stromu můžete vidět na výpisu příslušné tabulky v databázi (k výpisu jsem pro názornost použil pohled s odsazenými jmény). Sloupec `LEVEL` už samozřejmě chybí, neboť o hloubce jasně vypovídá délka genealogického řetězce.

```
select * from vw_tree;
```

ID	PADDED_NAME	VALUE	PARENT	GENEALOGICAL_ID
1	Moje hudba			A
2	Zpevaci	3	1	AA
3	Jarek Nohavica	3	2	AAA
4	Rok Dabla		3	AAAA
5	Mikymauzoleum		3	AAAB
6	Osma barva duhy		3	AAAC
7	Karel Kryl	2	2	AAB
8	Monology		7	AABA
9	To Nejlepsi		7	AABB
10	Manu Chao	1	2	AAC
11	Proxima Esperanza		10	AACA
12	Skupiny	2	1	AB
13	Schodiste	2	12	ABA
14	Mokry Pradlo		13	ABAA
15	Svinska Przola		13	ABAB
16	Hm..	3	12	ABB
17	Ehm		16	ABBA
18	To by mohlo byt zajimave		16	ABBB
19	Obed		16	ABBC

19 rows selected

Jako genealogický identifikátor zde byla zvolena velká písmena anglické abecedy. Je zřejmé, že 26 možností větvení nemusí vždy stačit, jistě ale není žádný problém místo jednoho písmene použít písmen několik. Někdy dokonce můžeme sáhnout po nějakém vhodném oddělovači (například znaku „/“) a uzly označovat čísly, každá sada sourozenců může mít variabilní délku identifikátoru.

Celá tato na první pohled složitá konstrukce slouží k obrovskému ulehčení při vyhledávání uzlů i větví, speciálně pak s různými omezeními. Veškeré tyto operace jsou díky této implementaci převedeny na jednoduchou a přehlednou práci s řetězci. Na druhou stranu, jisté nevýhody lze hledat v neefektivitě funkcí pro práci s řetězci a mírné obtíže při mazání i vkládání prvků.

5.1.3 Práce se sebereferenčními tabulkami

V úvodu této kapitoly je nejprve nutné stanovit základní operace, které budeme chtít se stromovými strukturami provádět. Mezi tyto operace řadíme:

- Vložení prvku
- Smazání prvku nebo celé větve stromu
- Výpis (chcete-li nalezení) prvku nebo větve stromu
- Hledání cesty ve stromu a to buď ke kořenu, nebo obecně mezi dvěma prvky

Důraz je samozřejmě kladen na poslední dvě operace, neboť je statisticky dokázáno, že výběry z dat jasně převažují nad úpravou dat. Navíc u vkládání a mazání prvků se asi nemá smysl bavit o nějaké optimalizaci, proto zde budou jen uvedena úskalí toho kterého rozšíření.

Vložení prvku

V případě vložení listu (uzlu bez potomků) jsou na tom všechny implementace sebereferenční tabulky víceméně stejně, respektive v případě rozšíření o následníky musíme vkládat do dvou tabulek najednou, což může vést ke znatelnému zpomalení při vkládání velkého množství prvků. Ani v případě rozšíření o genealogický identifikátor se nevyhneme určitým komplikacím – k jeho naplnění jsou potřeba dva (i když poměrně rychlé) dotazy `SELECT`, jeden pro výběr identifikátoru rodiče a druhý ke zjištění počtu následníků rodiče, abychom mohli vhodně doplnit identifikátor. Při hromadném vkládání si ale můžeme situaci zjednodušit (a urychlit) tím, že k počítání potomků využijeme proměnnou v PL/SQL proceduře. Popřípadě na to můžeme pamatovat již při implementaci a zavést si pro počet následníků další sloupec v tabulce.

```
-- zakladni sebereferencni tabulka
insert into tree values (526966, 'jmeno_uzlu', 5, 2);

-- se spojovaci tabulkou
insert into tree_w_connections values (142485, 'jmeno_uzlu', 5);
insert into tree_connections values (142484, 2, 142485);

-- s genealogickym identifikatorem
insert into tree_w_genealogical_id
values (461853,
       'jmeno_uzlu',
       5,
       2,
       (select genealogical_id from tree_w_genealogical_id where id = 2)
       || chr(65 + (select count(id) from tree_w_genealogical_id where parent=2))
       );
```

Pro srovnání: Při generování cvičných dat jsem zjistil, že v plnění základní sebereferenční tabulky a tabulky s genealogickým identifikátorem téměř není rozdíl. Za čas kolem 1 minuty se naplnilo do obou tabulek cca půl milionu řádek, zatímco do tabulky se spojovací tabulkou se za stejný čas naplnilo jen něco přes sto tisíc řádek.

Při vkládání „meziuzlu“ nastává (z hlediska rozšíření) situace zcela opačná. Máme-li spojovací tabulku, stačí jen vložit uzel a provést aktualizaci odkazů ve spojovací tabulce, podobně jako při základní implementaci, kde jen aktualizujeme odkaz na rodiče. Zato genealogický identifikátor musíme změnit v celé větvi, do které jsme nový uzel vložili – musíme tedy regenerovat identifikátory pro velké množství uzlů.

Vzhledem k tomu, že dotazy `INSERT` jsou stejné jako v prvním případě, uvádím jen dotazy pro aktualizaci následníků.

```
-- zakladni sebereferencni tabulka, tabulka s genealogickymi identifikatory
update tree set parent = ID_NOVEHO_UZLU where parent = ID_STAREHO_RODICE;

-- pro spojovaci tabulku
update tree_connections set parent = ID_NOVEHO_UZLU
  where parent = ID_STAREHO_RODICE;

-- pro tab s genealogickymi identifikatory navíc (pomala procedura)
exec pcg_genealogical_tree.generate_identifiers_recursive(ID_NOVEHO_UZLU);
```

Smazání prvku, větve

Mazání listů je ve všech implementacích víceméně totožné, jen u spojovací tabulky bude mazání opět probíhat ve dvou krocích.

Mazaní obecného prvku (tedy kdesi „uprostřed“ stromu) je stejný případ jako jeho vložení – zatímco u základní sebereferenční tabulky a tabulky s vazbami ve spojovací tabulce

nám postačí aktualizace záznamů (připojení na jiného, nadřazeného rodiče), v případě genealogických identifikátorů opět budeme nuceni identifikátory v postižené větvi znovu generovat.

Nejsložitějším problémem bude jistě mazání celé větve stromu. Ne však, máme-li k disposici genealogický identifikátor. V tomto případě totiž uplatníme obrovský komfort při práci s genealogickým identifikátorem. Pro smazání celé větve bude postačovat následující dotaz:

```
delete from tree_w_genealogical_id where genealogical_id like = ´AAA%´;
```

V případě ostatních implementací ovšem jeden dotaz rozhodně stačit nebude. Spokojme se tedy zatím s konstatováním, že jde o stejný úkol, jakým je vyhledání větve stromu, včetně všech jeho podřazených větví (tzv. podstrom). Tento úkol bude řešen několika způsoby v následující kapitole.

Vyhledávání ve stromu

Rekurze

Nejčastější operací, kterou budeme nad stromy provádět je výpis, respektive vyhledání stromu, nebo některé jeho větve. K dosažení tohoto úkolu se dá použít několika prostředků, z nichž nejznámější je rekurzivní procedura, která, zavolaná nad nějakým uzlem (zcela obecně, tedy i kořenem pro výpis celého stromu), najde všechny jeho potomky a pro každého potomka pak zavolá sama sebe a opět najde všechny jeho potomky, dokud tito potomci existují. Vývojový diagram je k nalezení v příloze I.

Tuto rekurzi můžeme obecně provádět na aplikační úrovni v jakémkoliv jazyce. Avšak nejlepší volbou bude jednoznačně PL/SQL. Nejenže budeme moci použít některé pokročilejší a rychlejší funkce, které Oracle nabízí (například dávkové zpracování – Bulk Collect), ale navíc budeme mít k disposici vnitřní tabulky Oracle namísto rekurzivního zásobníku ostatních jazyků. S vnitřními tabulkami se nemusíme bát přetečení zásobníku ani nedostatečné rychlosti, díky jejich optimalizaci. Skutečně, nepoužijeme-li některé rozšíření sebereferenčních tabulek a budeme-li data vybírat „přímo“, je rekurze ze všech způsobů nejrychlejší!

V následující tabulce jsou výsledky běhu rekurzivní procedury pro vyhledávání podstromů, která byla testována na velkém stromu s 526965 uzly s maximální hloubkou 10.

Procedura přijímá jeden argument, a sice uzel, jehož potomky hledáme, v případě testu byl tímto argumentem kořen, čili se jednalo o výpis celého stromu.

Tabulka 1 - Profil rekurzivní procedury pro nalezení větve stromu

Unit	Line	Total time	Occurrences	Text
PCG_TREE	84	1,760	526965	function get_ids_recursive(parent_id IN NUMBER) return t_ids_arr is
PCG_TREE	85	0,151	526965	ids t_ids_arr := t_ids_arr();
PCG_TREE	86	0,119	526965	tmp t_ids_arr := t_ids_arr();
PCG_TREE	87	0,055	526965	idx pls_integer :=1;
PCG_TREE	89	0,000	526965	begin
PCG_TREE	90	24,376	526965	select id bulk collect into ids from tree where parent = parent_id;
PCG_TREE	91	0,324	526965	idx := nvl (ids.count+1,1);
PCG_TREE	93	0,315	1053929	for i in 1..ids.count loop
PCG_TREE	94	0,338	526964	tmp := get_ids_recursive(ids(i));
PCG_TREE	95	0,092	1053928	if tmp is null then return null; end if;
PCG_TREE	97	0,676	5061754	for j in 1..tmp.count loop
PCG_TREE	98	2,163	4534790	ids.extend;
PCG_TREE	99	1,945	4534790	ids(idx) := tmp(j);
PCG_TREE	100	0,363	4534790	idx := idx + 1;
PCG_TREE	101	0,000	526964	end loop;
PCG_TREE	102	0,000	526965	end loop;
PCG_TREE	104	0,828	526965	return ids;
PCG_TREE	105	0,447	526965	end;
PCG_TREE	107	0,000	1	procedure print_ids(parent_id IN NUMBER) is
PCG_TREE	109	0,000	1	ids t_ids_arr := t_ids_arr();
PCG_TREE	112	0,000	1	begin
PCG_TREE	114	0,000	1	ids := get_ids_recursive(parent_id);
PCG_TREE	120	0,000	1	dbms_output.put_line('Nalezeno ' || ids.count || ' zaznamu.');
PCG_TREE	121	0,003	1	end;
TIME_TOTAL				43.578 seconds

Poznámky ke statistice: V tabulce můžeme vidět jednotlivé řádky kódu a jejich časovou náročnost. Lze z ní dobře vyčíst, že nejdelší čas stráví stroj hledáním potomků daného uzlu (řádek 90). Jak se později ukáže, je právě tento SQL dotaz alfou omegou nároku celé procedury.

Celkový čas 43,5 sekundy je dost zkreslený tím, že Oracle musel proceduru profilovat. Bez zapnutého profilování se reálný čas výpočtu pohyboval okolo 30 sekund.

Jen pro úplnost podotýkám, že to byl takzvaný warm-up run procedury – již předtím byla několikrát spuštěna, aby se při testu stroj nezabýval zbytečnými diskovými a paměťovými operacemi. Tento postup budu i nadále dodržovat při vytváření dalších statistik.

Nerekurzivní procedura

Rekurze není jediným řešením problému vyhledávání uzlů. Můžeme použít i dopřednou proceduru, která bude fungovat na bázi zásobníkového zpracovávání. Z hlediska PL/SQL ale máme tu výhodu, že namísto vlastní či již předpřipravené implementace bufferu raději použijeme rychlejší a hlavně „neomezenou“ dočasnou tabulku Oracle (temporary table). Opět také můžeme upotřebit některé specifické PL/SQL funkce.

Procedura funguje tak, že pro uzel, pro který jsme ji volali, najde potomky, které vloží do pomocné tabulky a ukazatele nastaví na začátek a konec nově vložených záznamů. Pro tyto záznamy pak v cyklu najde potomky daných uzlů a vkládá je do pomocné tabulky. Když cyklus skončí, přenastaví ukazatele znovu na nové záznamy a zopakuje cyklus. Vše probíhá tak dlouho, dokud se daří nacházet nové uzly. Celý princip tedy tkví v tom, že namísto rekurzivně volané funkce se používají dva cykly. V příloze J naleznete její vývojový diagram.

Tabulka 2 - Profil nerekurzivní funkce pro vyhledávání podstromů

Unit	Line	Total time	Occurrences	Text
PCG_TREE	49	0,000	1	begin
PCG_TREE	50	0,000	11	loop
PCG_TREE	51	0,098	526976	for i in lo_bound..hi_bound loop
PCG_TREE	53	46,544	526965	select id bulk collect into tmp_arr from tree where parent = (select val from tmp_ids where id = i)
PCG_TREE	56	19,032	674878	forall i in 1..tmp_arr.count insert into tmp_ids values (sq_tmp_ids.nextval, tmp_arr(i));
PCG_TREE	59	0,336	1201843	if tmp_arr.count > 0 then changed := true; end if;
PCG_TREE	60	0,184	526965	counter := counter + tmp_arr.count;
PCG_TREE	62	0,000	11	end loop;
PCG_TREE	64	0,000	22	if not changed then exit; end if;
PCG_TREE	67	0,000	10	lo_bound := hi_bound + 1;
PCG_TREE	68	0,000	10	hi_bound := hi_bound + counter;
PCG_TREE	69	0,000	10	changed := false;
PCG_TREE	70	0,000	10	counter := 0;
PCG_TREE	81	0,064	1	return ids;
PCG_TREE	82	0,005	1	end;
TIME_TOTAL				66.425 seconds

Poznámky ke statistice: Je dobře vidět, že dotaz pro vybírání potomků (řádek 53) je v tomto případě složitější, neboť musíme vybírat také z pomocné tabulky tmp_ids, dotaz je tedy časově náročnější.

Obě procedury by se daly jistě zjednodušit, a to přidáním podmínky, která by zajistila, že se nebudou vyhledávat potomci listů. To by přineslo značný nárůst rychlosti při vyhledávání ve velkých stromech (pro malé by byla další podmínka naopak zpomalující), avšak z hlediska porovnání obou procedur je toto vylepšení irelevantní – stále by byla dopředná procedura pomalejší kvůli složitějšímu dotazu pro vyhledání potomků.

Srovnání obou procedur se nabízí i v přílohách A a B, kde jsou další statistiky jejich běhu. Z nich je patrné, jak je v případě Oracle neefektivní řešení s pomocnou tabulkou (nahrazující tolik oblíbený buffer v jiných jazycích). Ačkoliv zatížení procesoru je skoro stejné, nerekurzivní procedura musí vrátit 2x více řádků než rekurze, to je dáno ukládáním hodnot právě do dočasné tabulky. Lepší je tedy spolehnout se na vnitřní zásobník Oracle, který je zcela jistě více zoptimalizován. Rekurze navíc nevykazuje ani žádné nároky na paměť, je to tím, že Oracle si záznamy o chodu ukládá do svých interních tabulek, které se do statistik nepromítnou.

Rekurzivní SQL

Oracle disponuje několika SQL funkcemi, pomocí kterých můžeme přímo pracovat s hierarchickými daty, která jsou uložena v sebereferenční tabulce. Podobné rozšíření SQL je také k disposici i v jiných systémech (například Postgre), ale v době psaní této práce šlo zatím jen o beta verze, které neměly tolik funkcí.

Základem práce s hierarchickými daty je klauzule `CONNECT BY PRIOR`, která specifikuje propojení rodič-potomek, respektive sloupec, který je odkazem do téže tabulky. Doplňující klauzule `START WITH` říká stroji, od kterého záznamu má začít. Následující dotaz vypíše celý strom.

```
select
  id        ID,
  name      Nazev,
  level     Hloubka,
  value     Hodnota,
  parent    Rodic
from tree
connect by prior id = parent
start with id = 1;
```

Již zde máme k disposici aparát, který nahradí procedury z předchozích odstavců, neboť v části `START WITH` si snadno vybereme požadovaný uzel, jehož podstrom chceme vypsat. Tím ovšem rekurzivní SQL v Oracle nekončí.

K disposici je totiž ještě několik šikovných funkcí i pseudosloupců, které zastanou spoustu práce. Jedním z nich je pseudosloupec `LEVEL`, který sám spočítá hloubku zanoření uzlu. Užitečné jsou funkce `CONNECT_BY_ISLEAF` a `CONNECT_BY_ISCYCLE`, které u každého záznamu zjistí, jestli se jedná o list, respektive jestli je uzel součástí cyklu. Zajímavý pseudosloupec je `SYS_CONNECT_BY_PATH`, který vrací cestu od kořene (tedy vlastně genealogický identifikátor). V následujícím příkladu, který demonstruje využití těchto funkcí je použita ještě klauzule `ORDER SIBLINGS BY`, která řadí potomky podle zadaného sloupce.

```
select
  id                                  ID,
  SYS_CONNECT_BY_PATH (name, '/')     "Cesta",
  level                               "Hloubka",
  parent                              "Rodic",
  case CONNECT_BY_ISLEAF
       when 1 then '   ANO'
       when 0 then '   NE'
  end                                 "Je list?"
from tree
connect by prior id = parent
start with id = 1
order siblings by name;
```

```
  ID Cesta                                                Hloubka Rodic Je list?
---- -------------------------------------------------- ------- ----- --------
   1 /Moje hudba                                              1          NE
  12 /Moje hudba/Skupiny                                      2     1    NE
  16 /Moje hudba/Skupiny/Hm..                                 3    12    NE
  17 /Moje hudba/Skupiny/Hm../Ehm                             4    16    ANO
  19 /Moje hudba/Skupiny/Hm../Obed                            4    16    ANO
  18 /Moje hudba/Skupiny/Hm../To by mohlo byt zajimave        4    16    ANO
  13 /Moje hudba/Skupiny/Schodiste                            3    12    NE
  14 /Moje hudba/Skupiny/Schodiste/Mokry Pradlo               4    13    ANO
  15 /Moje hudba/Skupiny/Schodiste/Svinska Przola             4    13    ANO
   2 /Moje hudba/Zpevaci                                      2     1    NE
   3 /Moje hudba/Zpevaci/Jarek Nohavica                       3     2    NE
   5 /Moje hudba/Zpevaci/Jarek Nohavica/Mikymauzoleum         4     3    ANO
   6 /Moje hudba/Zpevaci/Jarek Nohavica/Osma barva duhy       4     3    ANO
   4 /Moje hudba/Zpevaci/Jarek Nohavica/Rok Dabla             4     3    ANO
   7 /Moje hudba/Zpevaci/Karel Kryl                           3     2    NE
   8 /Moje hudba/Zpevaci/Karel Kryl/Monology                  4     7    ANO
   9 /Moje hudba/Zpevaci/Karel Kryl/To nejlepsi               4     7    ANO
  10 /Moje hudba/Zpevaci/Manu Chao                            3     2    NE
  11 /Moje hudba/Zpevaci/Manu Chao/Proxima Esperanza          4    10    ANO

19 rows selected
```

Tabulka 3 - Shrnutí rekurzivních SQL funkcí Oracle

CONNECT BY PRIOR	Podmínka, která definuje závislost mezi rodiči a jejich potomky
START WITH	Podmínka specifikující, který uzel bude brán ve výběru jako kořen (odkud tedy začne provádění dotazu)
LEVEL	Pseudosloupec vracející hloubku uzlu
CONNECT_BY_ISLEAF	Funkce, která vrací 1 v případě, že uzel je list a 0 v případě, že není
CONNECT_BY_ISCYCLE	Funkce, která vrací 1 v případě, že uzel je součástí cyklu
SYS_CONNECT_BY_PATH	Pseudosloupec, ve kterém je uložena cesta ke kořenu (resp. uzlu, který je vybrán v podmínce START WITH)
ORDER SIBLINGS BY	Zajistí řazení potomků podle sloupce
CONNECT_BY_ROOT	Pseudosloupec, ve kterém jsou uloženy hodnoty kořenu (resp. uzlu, který je vybrán v podmínce START WITH)
NOCYCLE	Specifikuje, že se má provádění příkazu zastavit, narazí-li SQL stroj na cyklus (a zastaví se bez chybového hlášení)

Syntaxe	`CONNECT BY PRIOR [NOCYCLE] <podmínka>`
	`START WITH <podmínka>`
	`ORDER SIBLINGS BY <sloupec>`

Zbývá tedy jen vyřešit otázku rychlosti, aneb jak je rekurzivní SQL rychlé? Při testu ve stejných podmínkách, v jakých běžely výše zmiňované procedury, vypsání stromu o 526965 uzlech trvalo **44,857 sekund**, což je výsledek, který se řadí mezi rekurzivní proceduru a její dopředný ekvivalent. Ale je nutné ihned dodat, že tento výsledek platí až pro takto obrovský strom! Pro stromy menší (až do 100000 uzlů) bylo rekurzivní SQL vždy spolehlivě rychlejší, jak dokládají srovnávací tabulky v příloze A. Také je třeba brát v úvahu nulové nároky na implementaci a hlavně pružnost, které nám tyto funkce poskytují – například takové setřídění výsledků podle některého sloupce by v případě procedur byl dost náročný problém!

> *Poznámka: V době psaní práce jsem nenašel ani zmínku o tomto nenadálém razantním zhoršení výkonu rekurzivního SQL. Zlom, který nastává pro stromy s více než 100000 uzly jsem určil experimentálně.*

Rozšíření sebereferenční tabulky

Z předchozího textu je zřejmě patrné, že pokud budeme potřebovat při vyhledávání opravdový nárůst výkonu (tedy za předpokladu, že nemáme k disposici rekurzivní SQL nebo chceme zpracovávat opravdu obrovské stromy), nevyhneme se implementaci některého rozšíření sebereferenční tabulky. Ideální pro případy vyhledání potomků, větví i celých podstromů je využít genealogický identifikátor.

S tímto rozšířením pak pro vypsání celého stromu (čili obdoba volání procedur a rekurzivního SQL z předchozích odstavců) postačí jednoduchý dotaz do databáze:

```
select * from tree_w_genealogical_id
  where genealogical_id like 'A%'
  order by genealogical_id;
```

Tento dotaz proběhl na mém počítači za **1,4 sekundy**, což je samozřejmě nesrovnatelně lepší výsledek než v případech procedur a rekurzivního SQL. Specifičtější výběry se pak dají realizovat právě přes genealogický identifikátor, k vypsání větve stromu postačí úplně stejný dotaz, jen v podmínce uvedeme jiný identifikátor. Jedinou větší nevýhodou je nemožnost abecedního setřídění výstupu. Záznamy bychom totiž museli abecedně už vkládat (například pomocí triggeru), jinak musíme samozřejmě řadit podle genealogického identifikátoru.

Toto rozšíření je, zdá se, obzvláště oblíbené u správců internetových diskusí a e-shopů, neboť je jednoduše implementovatelné i v jiných databázových serverech než v Oracle, navíc umožňuje lehce vypsat například jen přímé potomky uzlu (užitečné při rozbalování/sbalování vláken diskusí (produktů)). Ovšem i ostatní rozšíření základní implementace mají své výhody, o spojovací tabulce předchůdců a následníků se dozvíte v kapitole o síťových grafech, proto ji na tomto místě vynechám.

5.2 Nested sets

Pojem „nested sets“ zavedl americký programátor a SQL odborník Joe Celko, do češtiny by se dal přeložit jako „vnořené množiny“. Někdy se také používá názvu „DFS strom“, který je zkratkou anglického „depth first search”, což je název algoritmu, který při vytváření strom ohodnotí.

Principem je tedy specificky ohodnocená sebereferenční tabulka, která každému uzlu přiřadí dvě hodnoty, jejichž rozsahy značí, do které množiny uzly náleží. I když tedy i tato implementace stromu vychází ze sebereferenční tabulky, po jejím hodnocení už jsou údaje o rodiči zbytečné a práce s „nested sets” pak funguje jen na bázi ohodnocení, proto je uvádím v samostatné kapitole.

5.2.1 Implementace

Pro implementování „nested sets“ bude potřeba rozšířit sebereferenční tabulku o dva sloupce. Ve většině případů se používají označení „left“ a „right“, avšak vzhledem k tomu, že v Oracle jde o rezervovaná slova, použil jsem zkratky „lft“ a „rgt“.

```
create table tree_w_nested_sets
  (id        number not null primary key,
   name      nvarchar2(40),
   value     number,
   parent    number,
   lft       number,
   rgt       number
  );

alter table tree_w_nested_sets add constraint FK_tree_w_sets foreign key (parent)
references tree_w_nested_sets (id);
create index ix_tree_w_sets_parent on tree_w_nested_sets(parent);
create index ix_tree_w_sets_lft on tree_w_nested_sets(lft);
create index ix_tree_w_sets_rgt on tree_w_nested_sets(rgt);
```

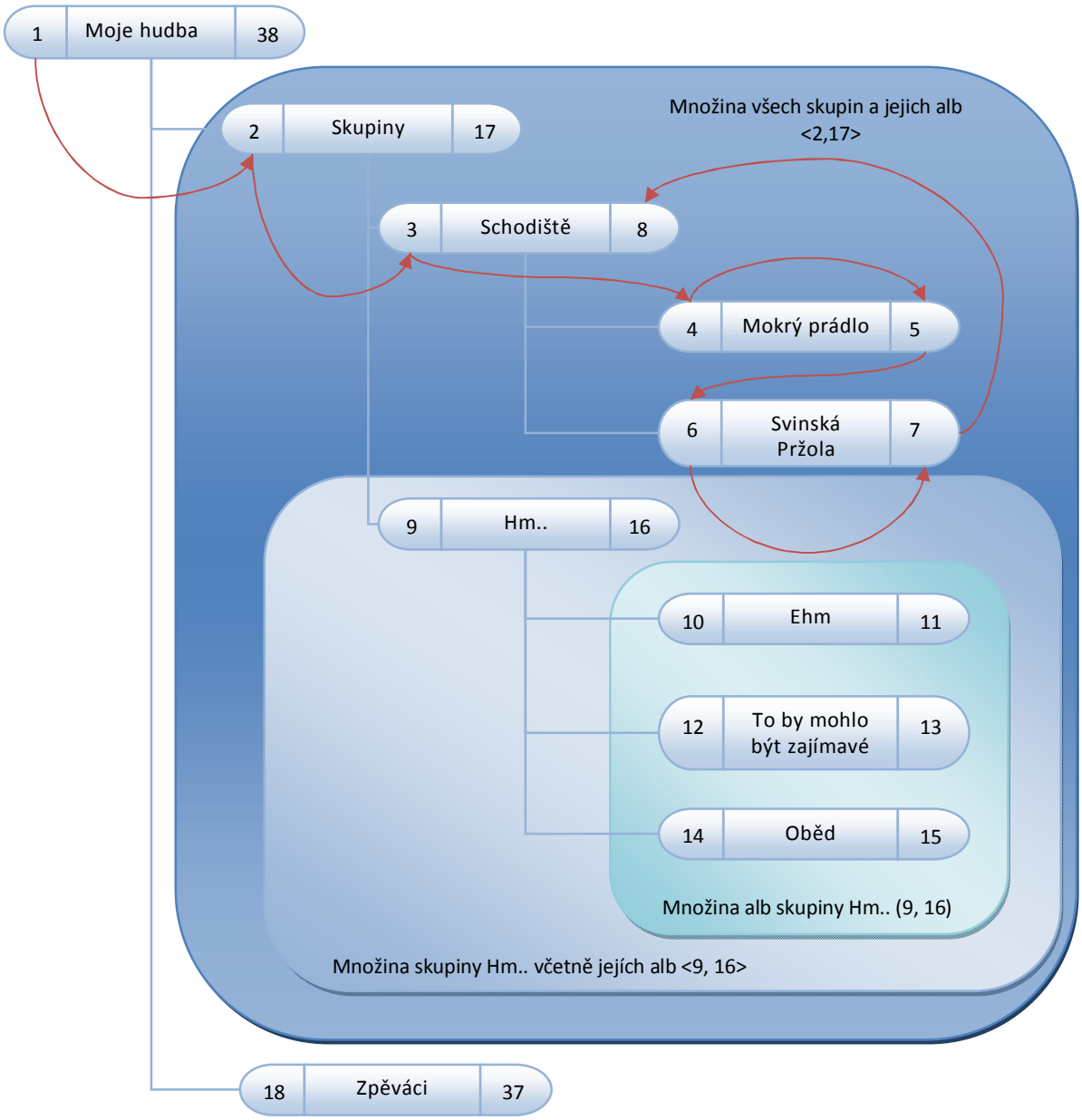
Poznámka: V tabulce TREE_W_NESTED_SETS budeme vyhledávat primárně podle sloupců LFT a RGT, proto nesmíme zapomenout vytvořit příslušné indexy!

Máme-li tabulku s hodnotami, zbývá ještě vygenerovat pomocí DFS algoritmu (někdy také „modified preorder tree traversal algorithm“) hodnoty pro sloupce `LFT` a `RGT`. Algoritmus se dá shrnout do několika kroků:

1) Nalézt uzel (pro který proceduru voláme) a zvednout počítadlo o 1.
2) Ohodnotit uzel.LFT = počítadlo.
3) Pro všechny potomky uzlu zavolat krok 1 (s parametrem ID potomka).
4) Zvednout počítadlo o 1 a přiřadit jeho hodnotu uzlu.RGT

Algoritmus se dá úspěšně implementovat pomocí rekurze, pro jeho lepší pochopení je znázorněn ve vývojovém diagramu v příloze K. Při jeho bližším prozkoumání, ke kterému nám poslouží i jeho grafické znázornění, zjistíme, že všichni potomci jednoho uzlu mají hodnoty LFT a RHT mezi stejnými hodnotami nadřazeného uzlu, odtud vyplývá název „vnořené množiny“ – čísla LFT a RGT totiž udávají, do které množiny ten který uzel patří.

Na obrázku 5 můžete vidět naznačení průběhu algoritmu (červené čáry) částí ukázkového stromu „moje hudba“, včetně ukázky některých množin s jejich číselnými intervaly. Za povšimnutí stojí fakt plynoucí z průběhu algoritmu, a sice že všechny listy mají hodnotu RGT o 1 větší než LFT, tedy můžeme tvrdit, že je-li RGT = LFT + 1, pak uzel je list.



Obrázek 5 - Výřez stromu "Moje hudba" s naznačením průbehu algoritmu a některých množin

Po proběhnutí algoritmu již dostáváme ohodnocený strom:

```
select
  lft                                          as LFT,
  '   ' || lpad(' ', (level - 1) * 2) || name as "Name",
  rgt                                          as RGT
from tree_w_nested_sets
connect by prior id = parent
start with id = 1;
```

```
       LFT Name                                    RGT
---------- ---------------------------------- ----------
         1    Moje hudba                             38
         2      Skupiny                              17
         3        Schodiste                           8
         4          Mokry Pradlo                      5
         6          Svinska Przola                    7
         9        Hm..                                16
        10          Ehm                               11
        12          To by mohlo byt zajimave          13
        14          Obed                              15
        18      Zpevaci                               37
        19        Jarek Nohavica                      26
        20          Rok Dabla                         21
        22          Mikymauzoleum                     23
        24          Osma barva duhy                   25
        27        Karel Kryl                          32
        28          Monology                          29
        30          To nejlepsi                       31
        33        Manu Chao                           36
        34          Proxima Esperanza                 35

19 rows selected
```

S generováním záznamů LFT a RGT algoritmem ovšem souvisí jedna poměrně nepříjemná skutečnost, a sice že je algoritmus poměrně dost pomalý.

Unit	Line	Total time	Occurrences	Text
PCG_NESTED_SETS	87	2,016	526965	procedure mark_nodes (node_id IN NUMBER) is
PCG_NESTED_SETS	91	0,000	526965	Begin
PCG_NESTED_SETS	92	0,000	526965	cnt := cnt + 1;
PCG_NESTED_SETS	93	90,093	526965	update tree_w_nested_sets set lft = counter where id = node_id;
PCG_NESTED_SETS	95	493,086	1580894	for child in (select id from tree_w_nested_sets where parent = node_id) loop
PCG_NESTED_SETS	96	0,820	526964	mark_node(child.id, counter);
PCG_NESTED_SETS	97	0,000	526965	end loop;
PCG_NESTED_SETS	100	107,390	526965	update nested_sets set rgt = cnt where id = node_id;
PCG_NESTED_SETS	101	8,537	526965	end;
PCG_NESTED_SETS	106	0,000	1	begin
PCG_NESTED_SETS	107	72,109	1	update tree_w_nested_sets set lft = -1, rgt = -1;
PCG_NESTED_SETS	108	0,000	1	mark_node(1,counter);
PCG_NESTED_SETS	109	0,000	1	end;
TIME_TOTAL				773.23 seconds

5.2.2 Práce s „nested sets“

Vložení prvku

V některých článcích se dočtete, že vložení prvku do „nested sets“ je podmíněno regenerováním hodnot LFT a RGT. Naštěstí je toto tvrzení mylné. Vložení uzlu s sebou sice přináší jistá úskalí, ale nic, s čím bychom si neuměli poradit. Nejprve je před samotným vkládáním provést `UPDATE` tabulky, kterým si pro prvek nejprve vytvoříme místo. Úplně stejně potom naložíme i s prvkem, jež listem být nemá, jak se můžete přesvědčit v ukázce nad stromem „moje hudba“.

```
update tree_w_nested_sets set
      lft = (case  when lft >= 10
                   then lft + 2
                   else lft
             end),
      rgt = (case  when rgt >= 10
                   then rgt + 2
                   else rgt
             end);

insert into tree_w_nested_sets values (20, 'Moje smutne srdce', 3, 10, 11);

update tree_w_nested_sets set
      lft = (case  when lft >= 31
                   then lft + 4
                   else lft
             end),
      rgt = (case  when rgt >= 31
                   then rgt + 4
                   else rgt
             end);

insert into tree_w_nested_sets values (21, 'Mnaga a Zdorp', 12, 31, 34);

insert into tree_w_nested_sets values (22, 'Nic Slozityho', 21, 32, 33);

select
  lft                                          as LFT,
  '   ' || lpad(' ', (level - 1) * 2) || name as "Name",
  rgt                                          as RGT
from tree_w_nested_sets
connect by prior id = parent
start with id = 1
order by lft;
```

```
       LFT Name                                    RGT
---------- ------------------------------------ ----------
         1    Moje hudba                            44
         2      Zpevaci                             23
         3        Jarek Nohavica                    12
         4          Rok Dabla                        5
         6          Mikymauzoleum                    7
         8          Osma barva duhy                  9
        10          Moje smutne srdce               11
```

```
        13         Karel Kryl                          18
        14           Monology                          15
        16           To nejlepsi                       17
        19         Manu Chao                           22
        20           Proxima Esperanza                 21
        24       Skupiny                               43
        25         Schodiste                           30
        26           Mokry Pradlo                      27
        28           Svinska Przola                    29
        31         Mnaga a Zdorp                       34
        32           Nic Slozityho                     33
        35         Hm..                                42
        36           Ehm                               37
        38           To by mohlo byt zajimave          39
        40           Obed                              41

22 rows selected
```

Pokud použitý databázový systém podporuje tabulkové spouště (triggery), jako například Oracle, můžeme `UPDATE` dotazy řešit právě pomocí triggerů a vkládání pak proběhne pomocí jednoduchého `INSERT` dotazu a my při vkládání nebudeme muset již nic ošetřovat. Jen se budeme muset i nadále smířit s delší odezvou na naše dotazy, neboť aktualizace velkého množství záznamů (v případě velkého stromu) není zrovna nejrychlejší SQL operací.

Smazání prvku, větve

Mazání prvku ani celé větve nám v této implementaci nezpůsobí žádné větší potíže – lze prostě požadované záznamy smazat. Pokud by nám z nějakého důvodu vadily mezery mezi hodnotami LFT a RGT, lze ještě aplikovat `UDPATE` dotazy podobně jako v případě vložení (nebo si pomoci triggerem), ale není to vůbec nutností.

Vyhledávání ve stromu

Vyhledávání ve stromu, výpis celých podstromů i nalezení cesty ke kořenu je hlavní zbraní „nested sets“. Různé jednoduché dotazy nám umožní dělat nad stromy všechny často prováděné operace s minimální časovou režií.

```
-- nalezeni podstromu skupiny „Hm..“
select id, lft, name, rgt from tree_w_nested_sets where lft >= 35 and rgt <= 42;

        ID        LFT NAME                                RGT
---------- ---------- ------------------------------ ----------
        16         35 Hm..                                   42
        17         36 Ehm                                    37
        18         38 To by mohlo byt zajimave               39
        19         40 Obed                                   41
```

```
-- nalezeni predchudcu skupiny „Hm..“
select id, lft, name, rgt from tree_w_nested_sets where lft <= 35 and rgt >= 42;

        ID        LFT NAME                                RGT
---------- ---------- ------------------------------ ----------
         1          1 Moje hudba                             44
        12         24 Skupiny                               43
        16         35 Hm..                                  42
```

Výběry ze stromu implementovaného pomocí „nested sets“ jsou vůbec nejrychlejší metodou, kterou jsem testoval – výběr celého stromu (velký testovací strom s 526965 uzly) proběhl za **0,76 sekundy**. Výpis celého stromu jsem samozřejmě realizoval dotazem:

```
select * from tree_w_nested_sets where lft >= 1 and rgt <= 1053930;
```

Poznámka: Vyhledávání v „nested sets“ implementaci je ještě rychlejší než ve stromech s genealogickým identifikátorem, neboť vyhledáváme podle 2 sloupců s indexovanými čísly namísto vyhledávání pomocí funkcí určených pro řetězce.

5.3 Vyhodnocení

Rozhodně nelze vynést nějaký bezvýhradný soud nad ukázanými implementacemi stromů v databázových systémech. Při přístupu k tomuto problému je třeba zvážit několik aspektů, z nichž nejdůležitější jsou:

1. Použitý databázový systém
2. Pravděpodobná velikost stromu (počet uzlů, maximální hloubka)
3. Operace, které nad stromy budou prováděny

Pro malé stromy, například nějaké soukromé databáze hudebních alb nebo filmů nám bohatě postačí základní implementace sebereferenční tabulky, kdy výpisy a hledání v databázi budeme realizovat pomocí rekurze. Implementace je jednoduchá a přehledná, přenositelná mezi databázovými systémy. Při použití databázového systému Oracle navíc můžeme s přehledem a obrovskou výhodou využít integrované funkce, a nemusíme se tak ani zabývat programováním vyhledávání ve stromu a jeho výpisem.

Pokud ovšem prioritním kritériem bude velký výkon v „košatějších“ stromech, budeme se muset poohlédnout po některém rozšíření. V Oracle se přímo nabízí využití kombinace genealogického identifikátoru zároveň s vnitřními funkcemi pro hierarchické

struktury, které jsou v mnohém užitečné a navíc pro nás mohou genealogické identifikátory přímo generovat (vzpomeňme na funkci `SYS_CONNECT_BY_PATH`), a to jak pro celý strom, tak pro jeho části (v případě změn – přidávání či odebírání uzlů). Toto řešení je opravdu doporučitelné v těch případech, kdy víme, že budeme zpracovávat obrovské stromy, nebo nebudeme mít k disposici rekurzivní SQL, které je v Oracle opravdu mocným nástrojem.

Ovšem existují i případy, kdy bude strom často měněn nebo aktualizován, například několikrát denně v databázi výrobků ve velkoskladech, apod. Pak své uplatnění naleznou i „nested sets“, kterým časté DML operace nečiní vážnější problémy, navíc zde ve velkých stromech uplatníme jejich obrovskou efektivitu při vyhledávání.

6 Síťové grafy

Síťové grafy jsou po stromech druhou nejrozšířenější aplikací teorie grafů. Tato obrovská rozšířenost plyne hlavně z faktu, že jde o matematické (a v případě potřeb i grafické) vyjádření činností prováděných v rámci nějakého projektu. Klíčové je slovo „nějakého“ – může jít totiž o projekty v obecném slova smyslu, vždyť plánování je dnes součástí snad každého oboru. Ani programování jako takové se v případě složitějších projektů neobejde bez podrobného plánování, které zajišťují právě síťové grafy. Jednou z nejznámějších aplikací pro plánování je třeba „Project“ od společnosti Microsoft.

Největší využití ale nalezly síťové grafy v oboru stavitelství. Zde analýza síťových grafů přináší nejen usnadnění, ale hlavně zefektivnění prováděných činností. Situace je zde ale komplikovanější než jinde, narážíme na několik problémů, například na souběžné provádění několika činností, které ale musejí být synchronizovány nejen z hlediska časového, ale i z hlediska místa, na kterém probíhají. Častým problémem jsou také smlouvy na určitou dobu, které musejí být plněny bez ohledu na předchozí činnosti, nebo i „malicherný“ problém nedodání materiálu. Proto asi nikoho nepřekvapí, že existují velmi pokročilé metody plánování stavebních projektů.

Dříve se nejvíce využívala jednoduchá metoda kritické cesty (CPM – critical path method), která pracuje s principem, že každá činnost může začít až tehdy, kdy je předcházející činnost ukončena. S pomocí fiktivních a čekacích činností se ošetřily některé stavy, kdy například dvě činnosti probíhaly současně a na obou pak byla závislá následující činnost. V takovém grafu se pak dala najít takzvaná kritická cesta, která byla de facto nejdelší cestou v grafu (ať už šlo o časovou nebo finanční náročnost). Ovšem tento jednoduchý systém brzy přestal pro účely stavebního projektování vyhovovat a objevily se pokročilejší metody.

Jednou z nich byla metoda BKN (Baukasten Netzpaltzung), do češtiny překládaná jako metoda stavebnicového síťového plánování. Ta přidala k metodě kritické cesty navíc dělení vazeb činností (tedy vyjádření jejich návazností) a to 4 typy, ve kterých je definován minimální časový odstup, se kterým mohou činnosti začínat nebo končit.

Přesto se ale ukázalo, že ani metoda BKN není zcela univerzální (detaily přenechejme příslušným odborníkům) a vznikly i další, komplexnější řešení. Jedním z nich je STSG, neboli stavebně technologický síťový graf, metoda vyvinutá v Čechách profesorem Jarským. Toto řešení rozšiřuje BKN o další 4 vazby, pomocí kterých řeší nedostatky metody BKN.

Se síťovými grafy souvisí pojem *časové analýzy*, která je základem operací prováděných se síťovými grafy. Časová analýza je termín, který se používá pro vyjádření postupného ohodnocení síťových grafů tak, abychom získali na základě jeho procházení užitečné informace o časových termínech a rezervách, které následně hrají zásadní roli v plánování projektu. S touto analýzou souvisí tedy několik pojmů (veličin):

1. Nejdříve možný začátek činnosti
2. Nejdříve možný konec činnosti
3. Nejpozději přípustný začátek činnosti
4. Nejpozději přípustný konec činnosti

Jejich kombinací pak můžeme získat několik dalších údajů. Pro nás je ovšem důležité vědět, že získání právě těchto hodnot je náš primární úkol při zpracování síťového grafu.

6.1 Firemní zadání

Součástí zadání diplomové práce je požadavek na zpracování zmíněné metody stavebně technologických síťových grafů. Proto se v dalším textu budu zaobírat právě touto metodou, což ovšem není na škodu, neboť je v porovnání s ostatními metodami, které jsem jmenoval tou nejpokročilejší a nejnáročnější na zpracování, zahrnuje tedy i práci se staršími, jednoduššími metodami.

6.2 Pohled programátora

Jak je řečeno v teoretické části práce, „nejpříbuznější“ grafovou strukturou k síťovým grafům jsou stromy, proto asi nepřekvapí, že implementace síťových grafů nebude příliš odlišná. Zcela zásadní rozdíl musíme totiž hledat v požadavcích na prováděné operace, tedy v práci se síťovými grafy. Zde jsou odlišnosti bohužel dosti značné – nebudeme ani tak potřebovat síťové grafy vypisovat a vyhledávat v nich podgrafy, ale naopak v rámci časové analýzy musíme graf procházet oběma směry (tedy od počátečního uzlu ke koncovému a zpět) a při průchodech počítat veličiny související s časovou analýzou síťového grafu.

Čistě prakticky vzato, budeme muset „navštívit“ každý uzel grafu a na něm provést nějaké výpočty. A budeme muset navíc zachovávat určitou posloupnost, se kterou budeme uzly vybírat. Po provedení všech operací v jednom směru tento otočíme. Navíc je nutné tyto operace provádět co nejrychleji – funkce budou muset být volány při jakékoliv změně síťového grafu.

A ještě jedna poznámka: v odborné literatuře se můžete dočíst, že síťové grafy se liší podle toho, zda jsou činnosti zaváděné do grafu znázorněny uzly nebo vazbami. Z programátorského hlediska je to opět jedno, ať tak či onak, zavedeme si jednu tabulku pro uzly, druhou pro vazby a z hlediska práce s nimi není důležité, která tabulka je nositelem dat o činnostech… Přesto pro úplnost – při zpracování metodou STSG musí být činnosti znázorněné uzly, takže situace kolem implementace je pro nás přehlednější, a tedy i příznivější.

6.3 Implementace

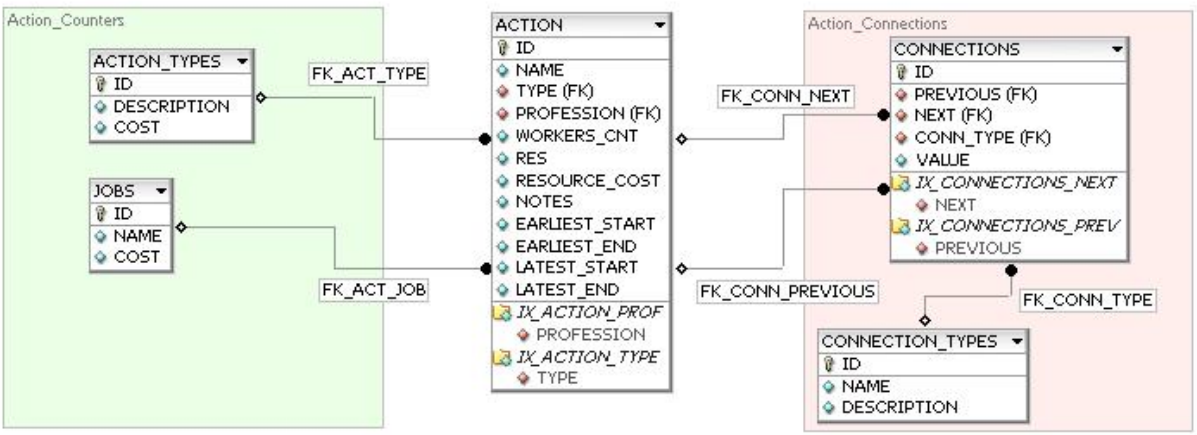
Implementace síťového grafu v databázi naštěstí není nic složitého. Když si uvědomíme hlavní rozdíl oproti stromům, tedy že uzel síťového grafu může mít více rodičů, lehce usoudíme, že ideální a vlastně i jedinou možnou implementací bude použití dvou tabulek. Jedna z nich bude reprezentovat uzly, činnosti, druhá naopak vazby. Jedná se vlastně o databázovou implementaci vazby N:N, s tím, že zde vazební tabulka ponese ještě další informace, například o druhu vazby, kterou zastupuje.

DDL příkazy pro vytvoření tabulky činností a jejich vazeb by mohly vypadat nějak takto:

```
create table action
  (id              number not null primary key,
   name            nvarchar2(40),
   type            number,
   profession      number,
   workers_cnt     number,
   res             nvarchar2(100),
   resource_cost   number,
   notes           nvarchar2(1000),
   earliest_start  number,
   earliest_end    number,
   latest_start    number,
   latest_end      number,
   duration_time   number
  );

alter table action add constraint FK_act_job foreign key (profession) references
jobs (id);
alter table action add constraint FK_act_type foreign key (type) references
action_types (id);


create index ix_action_type on action(type);
create index ix_action_prof on action(profession);

create table connections
  (id        number not null primary key,
   previous  number,
   next      number,
```

```
    conn_type number,
    value     number
  );

alter table connections add constraint FK_conn_type foreign key (conn_type)
references connection_types (id);



alter table connections add constraint FK_conn_previous foreign key (previous)
references action (id);
alter table connections add constraint FK_conn_next foreign key (next) references
action (id);



create index ix_connections_prev on connections(previous);
create index ix_connections_next on connections(next);
```

Poznámka: V rámci úspory místa jsem neuvedl dotazy pro vytvoření číselníkových tabulek „professions“, „action_types“ a „conn_types“. Pro další úvahy je jich netřeba, spokojme se jen s konstatováním, že například na druhu vazby závisí výpočet nejdřívějších startů a konců činností (a samozřejmě i nejpozdějších), ovšem vystačíme si s čísly odkazujícími do tabulky conn_types (abychom v podmínkách podle nich větvili logiku programu), nemusíme nutně znát názvy vazeb.



Obrázek 6 - databázový model implementace síťového grafu

Na obrázku můžete vidět kompletní model implementace síťového grafu včetně jeho číselníků. Model byl vygenerován pomocí aplikace DBDesigner 4.

6.4 Práce se síťovými grafy

Hlavní úkol, který před námi stojí při práci se síťovými grafy je vypočítání veličin časové analýzy, tedy spočítání nejdřívějších a nejpozdějších začátků i konců činností a to pro každou činnost. Hodnoty nejdřívějších začátků a konců získáme průchodem síťového grafu vpřed. Průchodem grafu vzad získáme hodnoty nejpozdějších začátků a konců. Je třeba připomenout, že hodnoty počítané při průchodu navíc závisejí na hodnotě a typu (sloupce „value“ a „conn_type“) příslušné vazby. Možnosti, které kombinacemi druhů vazeb a jejich hodnot mohou vzniknout, jsou mimo oblast našeho zájmu, postačí informace, že je třeba při výpočtu program ještě členit, což ve všech popisovaných případech zajistí poměrně rychlý SQL výraz `CASE`.

Obecné kroky pro generování potřebných hodnot by se daly shrnout takto, na některé změny, které jednotlivá řešení přinášejí, upozorním v příslušném odstavci:

1. Nastavit počátku hodnotu nejdřívějšího začátku na 0, nejdřívějšího konce na dobu činnosti (pokud jediný počátek nemáme, vytvoříme fiktivní a dobu procesu nastavíme na 0)
2. Nyní již v cyklu můžeme procházet činnosti a pro každou nastavit:
   nejdřívější start = max (nejdřívější konec předcházející činnosti),
   nejdřívější konec = nejdřívější start činnosti + doba trvání činnosti (sloupec duration_time)
3. Nastavit konci největší hodnotu ze všech možných nejdřívějších konců
4. Procházet činnosti odzadu, pro každou nastavit:
   nejpozdější konec = min (nejpozdější konec následující činnosti – délka následující činnosti),
   nejpozdější start = nejpozdější konec – doba trvání činnosti

Kroky tohoto algoritmu jsou čistě obecné, odpovídají průchodu při počítání metodou CPM. Kritickou cestu jsme takto skutečně zjistili, neboť kritické cesty budou mít stejné hodnoty u nejdřívějších a nejpozdějších konců. Při použití metody STSG tedy musíme navíc větvit výpočty v krocích 2 a 4 podle druhu vazby, kterou na sebe dvě činnosti navazují.

6.4.1 Základní PL/SQL procedura

První způsob, kterým jsme problém analýzy síťových grafů řešili, odpovídá 4 obecným krokům popsaným v předchozí kapitole (6.4) s tím, že procedura využívá hlavně PL/SQL. Vzhledem k tomu, že ve výpočetním kroku je nutné udělat dva oddělené úkony, vypočíst nejdřívější start a z něho následně vypočítat i nejdřívější konec, byli jsme nuceni použít `FOR` cyklus namísto rychlejšího `FORALL` cyklu. Podrobněji rozepsané kroky, které procedura provádí, vypadají takto:

1) Zjištění počtu počátků (odpovídá spočítání uzlů, které nemají předchůdce definované v tabulce „CONNECTIONS“), pokud jich je víc, přidat fiktivní, jedinečný počátek.
2) Vynulování záznamů o nejdřívějších startech a koncích všech uzlů, nastavení nejdřívějšího startu počátku na 0.
3) Ve `FOR` cyklu procházet uzly – hledat následníky již spočítaných uzlů a počítat pro ně hodnoty nejdřívějších startů. Z nich pak lze spočítat i nejdřívější konce. Samotný výpočet probíhá v SQL `UPDATE`, včetně dělení podle vazeb činností, které zajišťuje SQL `CASE`.
4) Zjištění počtu koncových uzlů, pokud jich je více, vytvořit fiktivní, jedinečný konec.
5) Nastavení maximální hodnotu ze všech možných nejdřívějších konců činností koncovému uzlu.
6) Ve `FOR` cyklu procházet uzly – hledat předchůdce již spočítaných uzlů a počítat hodnoty nejpozdějších konců. Z nich pak dopočítat nejpozdější starty.
7) Smazat vytvořené uzly (fiktivní počátek a konec), pokud existují.

Kromě kroků se můžete také orientovat podle vývojového diagramu v příloze L. Lehké vylepšení procedury je možné, pokud máme k disposici indexy, které reprezentují upořádání uzlů, což je ostatně všeobecně doporučováno. Toto vylepšení je použito i v naší proceduře, i v případě 3. řešení, popsaném v kapitole 6.4.3. Změní se kroky 3 a 6, ve kterých nemusíme vyhledávat uzly s již vyplněnými předchůdci, ale brát uzly postupně podle indexu.

V následující tabulce předkládám profil této procedury, která měla za úkol zpracovat malý projekt, stavbu rodinného domu, který zahrnuje 248 činností a 1163 vazeb těchto činností.

Tabulka 4 - Profil základní PL/SQL procedury pro časovou analýzu síťového grafu

Unit	Line	Total time	Occurrences	Text
PCG_GRAPH	12	0,000	1	function count_project_endpoints (project_id in number) return number is
PCG_GRAPH	14	0,000	1	Begin
PCG_GRAPH	15	0,006	1	select count(id) into cnt from action
PCG_GRAPH	18	0,000	1	return cnt;
PCG_GRAPH	19	0,000	1	end;
PCG_GRAPH	56	0,000	1	procedure calculate_project_ff (project_id IN NUMBER) is
PCG_GRAPH	59	0,000	1	Begin
PCG_GRAPH	64	0,031	1	update action set earliest_start = 0;
PCG_GRAPH	74	0,026	250	for cinnost in (select * from action order by id) loop
PCG_GRAPH	77	0,865	248	select max (z.earliest_start + z.duration_time) into v_max from action z
PCG_GRAPH	81	0,443	248	update action set earliest_start = nvl(v_max,0)
PCG_GRAPH	84	0,000	1	end loop;
PCG_GRAPH	86	0,000	1	end;
PCG_GRAPH	91	0,000	1	procedure calculate_project_rr (project_id IN NUMBER) is
PCG_GRAPH	94	0,000	1	fake_end boolean := false;
PCG_GRAPH	95	0,000	1	Begin
PCG_GRAPH	99	0,000	1	if (count_project_endpoints(project_id) > 1) then
PCG_GRAPH	102	0,000	1	end if;
PCG_GRAPH	105	0,014	1	update action set latest_end= 0;
PCG_GRAPH	115	0,027	250	for act in (select * from action order by idx desc) loop
PCG_GRAPH	118	0,625	248	select max (z.earliest_start + z.duration_time) into v_max from action z
PCG_GRAPH	122	0,390	248	update action set latest_end = nvl(v_max,0)
PCG_GRAPH	125	0,000	1	end loop;
PCG_GRAPH	127	0,000	1	if (fake_end) then
PCG_GRAPH	129	0,000	1	end if;
PCG_GRAPH	131	0,000	1	end;
TIME_TOTAL				2.427 seconds

Možná to není na první pohled zřejmé, ale časová náročnost procedury je neúnosně velká. Už při zpracování nejmenšího projektu, který jsem měl k disposici, trvá propočítání 248 činností téměř 2,5 sekundy. Při zpracování obrovského projektu, který má několik tisíc činností (a desítky tisíc vazeb) jsem se nedobral výsledku ani po více než minutě. A přitom je nutné spouštět tento výpočet při jakékoliv změně v grafu…

Abychom mohli proceduru vylepšit, bylo nutné analyzovat profil procedury. V tabulce je ovšem dobře vidět kámen úrazu tohoto řešení. Je jím cyklus `FOR`. Smyčka totiž nutí Oracle přepínat kontext mezi stroji pro zpracování PL/SQL cyklu a SQL dotazů. V každé iteraci cyklu musí databázový systém nejprve zvětšit iteraci (to provádí PL/SQL stroj), následně přepnout kontext na SQL stroj, který zpracuje dotaz `SELECT` (řádek - line 77), pak se kontext přepne znovu na PL/SQL stroj, který pouze zjistí, že má opět provést SQL dotaz (`UPDATE`,

řádek 81), a předá tak znovu kontext SQL stroji. To má za následek, že obyčejný `UPDATE` 248 záznamů trvá bezmála půl sekundy.

Další úkol zněl tedy jasně – přenést zátěž na SQL stroj a vyhnout se tak cyklu `FOR`.

6.4.2 SQL řešení

Klasickým řešením problému z minulé kapitoly je využití dávkového zpracování, čili cyklu `FORALL`. Ten nejprve připraví dávku SQL dotazů, které najednou odešle SQL stroji ke zpracování, ke změně kontextu dojde tedy jen jednou. Na druhou stranu ale umožňuje do cyklu vložit pouze jeden SQL dotaz, kdežto my máme dotazy dva. To by sice šlo obejít dvěma po sobě následujícími cykly, ale my jsme přišli ještě na jiné, rychlejší řešení.

Jeho pozadí je úzce spjaté s teorií grafů. Využili jsme faktu, že každý acyklický graf lze uspořádat – očíslovat tak, že vždy platí, že předchůdci mají vždy nižší index než následníci. Toho lze s výhodou využít k vytvoření dvou dotazů, které spočítají hodnoty pro celý graf a to vše bez použití cyklu. V ukázce je základ SQL dotazů, ovšem bez podrobností (samotné počítání nejdřívějších startů podle druhu vazby), pro průchod síťovým grafem vpřed.

```
update action act set
      act.earliest_start =
      (select
         max (
           case conn.connection_type
             when 1 then …
             when 2 then …
             when 3 then …

             when 4 then …
             when 5 then …
             when 6 then …
             when 7 then …
             when 8 then …
           end
       from connections conn
       join action prev
         on conn.previous = prev.id)
where conn.next = act.id;

update action act
set act.earliest_end = act.earliest_start + act.duration_time;
```

Poznámka: Data v tabulce action musí být uspořádána. Naše testovací data již naštěstí uspořádána byla, takže jsme se tímto problémem nemuseli zabývat.

Zrychlení celého procesu výpočtu veličin časové analýzy je úctyhodné, jak ukazuje tabulka 6. V ní je profil procedury, která zapouzdřuje SQL dotazy. Vidíme, že každý SQL dotaz proběhne na rozdíl od minulého řešení pouze jednou. Další dobrou vlastností našeho druhého řešení je i to, že již není třeba vytvářet fiktivní začátky a konce – v uspořádané tabulce nemůže dojít k situaci, kdy bychom chtěli měnit uzel, jehož předchůdce ještě nebyl zpracován.

Tabulka 5 - Profil procedury založené na SQL dotazu pro časovou analýzu síťového grafu

Unit	Line	Total time	Occurrences	Text
PCG_GRAPH	13	0,000	1	procedure calculate_project_ff (project_id IN NUMBER) is
PCG_GRAPH	19	0,000	1	begin
PCG_GRAPH	24	0,021	1	update action set earliest_start = 0, earliest_end= 0;
PCG_GRAPH	117	0,679	1	update action act set …
PCG_GRAPH	203	0,012	1	update action act set act.earliest_end = act.earliest_start + act.duration_time
PCG_GRAPH	209	0,000	1	update action set latest_end = (select max(earliest_end) from action)
PCG_GRAPH	214	0,586	1	update action act set …
PCG_GRAPH	300	0,004	1	Update action act set act.latest_start = act.latest_end – act.duration_time
PCG_GRAPH	307	0,000	1	end;
TIME_TOTAL				1.302 seconds

Ovšem ani tato procedura, respektive SQL dotaz nebyl přesvědčivě rychlý pro větší objem dat. Projekt s 5600 činnostmi spojených 26857 vazbami byl zpracován za **52 sekund**. Proto jsme pořád nebyli spokojeni a museli jsme výpočet ještě zrychlit.

6.4.3 Řešení s pomocí pohledu

Trik posledního řešení, které je již připraveno pro nasazení v cílové aplikaci, spočívá v přenesení výpočtů do pohledů. Jednoduše řečeno, dotaz, který zajišťuje samotný výpočet (je to víceméně stejný dotaz, jako ten v předchozí kapitole), již není volán v proceduře, která má zajistit propočítání časové analýzy, ale byl použit pro vytvoření pohledů. Jeden z nich je pohledem na data, která odpovídají průchodu vpřed, druhý naopak odpovídá průchodu uzly zpět. Do těchto pohledů se pak ve výpočetní proceduře dotazujeme.

Při změnách prováděných v síťovém grafu se pohledy vždy ukazují na příslušná data, ale tyto operace nezdržují samotné vkládání/mazání uzlů. Do pohledů, značně omezených výběrem jedno uzlu se pak dotazujeme v těle procedury. Algoritmus je podobný oběma předchozím (i když každému jinak), jeho grafické znázornění je v příloze M.

Nárůst výkonu je ještě markantnější než při porovnání prvních dvou řešení. O použitelnosti této metody svědčí i to, že projekt s 5600 činnostmi byl zpracován za **9 sekund**. Samotným vyplněním údajů odpovídají řádky 95, 104 a 138.

Tabulka 6 - Profil procedury s využitím dotazů do pohledu

Unit	Line	Total time	Occurrences	Text
PCG_GRAPH	28	0,000	1	procedure calculate_project(project_id IN NUMBER) is
PCG_GRAPH	33	0,000	1	begin
PCG_GRAPH	36	0,081	1	update action c
PCG_GRAPH	53	0,000	1	update action act
PCG_GRAPH	88	0,000	250	for c_akt in (select *
PCG_GRAPH	95	0,244	248	update action act
PCG_GRAPH	104	0,117	248	update action act
PCG_GRAPH	109	0,000	1	end loop;
PCG_GRAPH	113	0,036	1	select max(c.earliest_end) into max_konec from action c
PCG_GRAPH	115	0,010	250	for c_act in (select *
PCG_GRAPH	127	0,005	248	select min(vaz.new_kp) into nejp_konec from sv_sg_new_kp vaz where vaz.previous = c_act.id;
PCG_GRAPH	132	0,000	248	if nejp_konec is null
PCG_GRAPH	133	0,000	1	then nejp_konec := max_konec;
PCG_GRAPH	134	0,000	248	end if;
PCG_GRAPH	135	0,000	248	end if;
PCG_GRAPH	136	0,000	248	end if;
PCG_GRAPH	138	0,066	248	update action c
PCG_GRAPH	144	0,000	1	end loop;
PCG_GRAPH	146	0,000	1	end;
TIME_TOTAL				0.39 seconds

V příloze F jsou další statistické informace o běhu procedury. Porovnáme-li je se statistikami v příloze E, zjistíme, že hlavní rozdíl tkví v počtu řádků, které musel dotaz načíst (table scan rows gotten). S tím pak souvisí i větší nároky SQL řešení na fyzický přístup na disk, což je samozřejmě ten hlavní důvod zpoždění oproti pohledu. Z pohledu se každou chvíli vybírají jen data potřebná pro jeden uzel, kdežto SQL dotaz v proceduře vždy vyhledává v celé tabulce. O tom vypovídá počet řádků vrácených SQL strojem podle identifikátoru ROWID.

6.5 Vyhodnocení

Více informací nalezne čtenář v příloze H, ve které je srovnávací tabulka pro jednotlivé metody v závislosti na velikosti projektu. Z ní je patrné, že jsme dokázali úspěšně implementovat síťové grafy do databáze a následně dostatečně efektivně provést jejich časovou analýzu pro různě veliké projekty, které bude aplikace i v praktickém nasazení počítat. Poslední zmiňované řešení bude implementováno do cílové aplikace.

Na rozdíl od stromů, v případě síťových grafů existuje jen jedna možná implementace a navíc i jedno obecně doporučitelné řešení výpočtů časové analýzy. Samozřejmě se jedná o využití pohledů, ve kterých jsou předpřipravená a vždy aktualizovaná data. Pokud by ovšem programátor musel využít jiný databázový systém, který pohledy nepodporuje, nebo by z nějakého důvodu musel provádět časovou analýzu na aplikační úrovni, má možnost implementovat první řešení, které je čistě procedurální a dobře přenositelné i mezi ostatní programovací jazyky.

7 Závěr

V práci je nejprve shrnuta problematika teorie grafů, ve které jsou definovány termíny použité v další části práce. Toto shrnutí může být užitečné zejména programátorům, kteří v něm najdou jen termíny podstatné pro práci se stromy a síťovými grafy, a nemusí tak studovat odbornou literaturu, většinou určenou spíše pro matematiky.

V kapitole pojednávající o stromech jsou přehledně předvedeny různé možnosti implementace a práce se stromovými strukturami v prostředí relačních databází. Čtenáři zde najdou nejen ukázky implementací stromů, ale zároveň několik algoritmů, které jsou v databázovém prostředí Oracle testovány hlavně z hlediska jejich časové náročnosti. Dále je zde ukázka práce s rekurzivními dotazy, což jsou specifické funkce databáze Oracle. Nechybí ani jejich porovnání z hlediska náročnosti. Na konci kapitoly jsou pak shrnuta doporučení pro různé případy na základě informací získaných z testů různých přístupů ke stromům.

Poslední kapitola je věnována řešení problematiky síťových grafů v databázích. Tato kapitola pojednává o řešeních, která byla navržena pro využití v projektu zadávající firmy. Jejich postupným vývojem jsme došli k efektivnímu řešení, jež je připraveno k využití pro komerční účely a bude nasazeno v cílové aplikaci. Stejně jako v předchozích případech u stromů, i zde jsou jednotlivé přístupy testovány, hlavně z hlediska rychlosti, neboť vysoký výkon byl hlavním kritériem zadavatele. Proto si dovolím tvrdit, že jsem hlavní cíle práce splnil.

8 Použitá literatura

[1] Berka Milan: Operační výzkum, [online], cit.[28.11.2007], URL: < http://home.eunet.cz/berka/o/grafy.htm>

[2] Celko Joe: Trees In SQL, [online], Intelligent Enterprise Magazine, cit. [22.2.2008], URL: < http://www.intelligententerprise.com/001020/celko.jhtml?_requestid=1266295>

[3] Demel Jiří: Grafy

[4] Greenspun Philip: Representing Trees in Oracle SQL, [online], cit. [1.3.2008] URL: < http://www.abclinuxu.cz/clanky/navody/stromy-v-sql>

[5] Harrison Guy: Tuning Oracle PL/SQL, [online], fortunecity.com, cit. [5.4.2008], URL: < http://www.fortunecity.com/skyscraper/oracle/699/orahtml/ioug/tunplsql.html>

[6] Hillyer Mike: Managing Hierarchical Data in MySQL, [online], MySQL AB © 1995-2008, cit. [22.2.2008], URL: < http://dev.mysql.com/tech-resources/articles/hierarchical-data.html>

[7] Lacko Luboslav: Oracle Správa, programování a použití databázového serveru, 1. Vydání, Brno 2007

[8] Oracle 10g Release 2 (10.2) Documentation, [online], oracle.com 2008, cit. [5.12.2007], URL: < http://www.oracle.com/technology/documentation/database10gR2.html>

[9] Oracle Connect By, Version 1.11 [online], psoug.org, cit. [1.3.2008], URL: < http://www.psoug.org/reference/connectby.html>

[10] Ryjáček Z.: Teorie Grafů a diskrétní optimalizace 1, [online], Západočeská univerzita v Plzni 2007, cit. [30.11.2007], URL: < http://www.kma.zcu.cz/TGD1>

[11] Stěhule Pavel: Stromy, [online], root.cz 2005, cit. [14.2.2008], URL: < http://www.root.cz/clanky/stromy/>

[12] Szalbot Pavel: Stromy v SQL, [online], ABC Linuxu 2006, cit. [14.2.2008], URL: < http://www.abclinuxu.cz/clanky/navody/stromy-v-sql>

[13] Tropaskho Vadim: Trees in SQL: Nested Sets and Materialized Path, [online], DBAzine.com 2005, cit. [16.2.2008], URL: < http://www.dbazine.com/oracle/or-articles/tropashko4>

[14] Univerzitní e-learningový systém, [online], cit. [22.01.2008], URL: <http://e-learning.tul.cz/cgi-bin/elearning/elearning.fcgi>

[15] Urman Scott, Hardman Ron, McLaughlin Michael: Oracle Programování v PL/SQL, 1. vydání, Computer Press, Brno 2007

[16] Zelenka Petr: Metody ukládání stromových dat v relačních databázích, [online], interval.cz 2005, cit. [14.2.2008], URL: < http://interval.cz/clanky/metody-ukladani-stromovych-dat-v-relacnich-databazich/>

[17] KnowledgeBase firmy StringData, s.r.o.

9 Přílohy

Seznam příloh

Příloha A

Statistiky rekurzivní procedury pro procházení stromu

Name	Last
CPU used by this session	3368
CPU used when call started	3364
index fast full scans (direct read)	0
index fast full scans (full)	0
index fast full scans (rowid ranges)	0
physical read bytes	31776768
physical reads	3879
physical reads cache	3879
recursive calls	527710
recursive cpu usage	1417
redo blocks written	0
redo entries	143
redo size	104124
session logical reads	1755793
session pga memory	0
session stored procedure space	0
session uga memory	0
table fetch by rowid	526964
table scan rows gotten	0

Příloha B

Statistiky dopředné procedury pro procházení stromu

Name	Last
CPU used by this session	3833
CPU used when call started	3823
index fast full scans (direct read)	0
index fast full scans (full)	0
index fast full scans (rowid ranges)	0
physical read bytes	188399616
physical reads	22998
physical reads cache	22998
recursive calls	685656
recursive cpu usage	3299
redo entries	307978
redo size	58033084
session logical reads	4278232
session pga memory	65536
session stored procedure space	0
session uga memory	0
table fetch by rowid	1054178
table scan rows gotten	530777

Příloha C

Statistiky procedury pro označení „Nested Sets“

Name	Last
CPU used by this session	66018
CPU used when call started	66010
index fast full scans (direct read)	0
index fast full scans (full)	0
index fast full scans (rowid ranges)	0
physical read bytes	193101824
physical reads	23572
physical reads cache	23572
recursive calls	3192763
recursive cpu usage	20351
redo entries	5822438
redo size	1258839928
session logical reads	20794902
session pga memory	393216
session stored procedure space	0
session uga memory	196392
table fetch by rowid	527028
table scan rows gotten	562024

Příloha D

Statistiky základní PL/SQL funkce pro procházení síťového grafu

Name	Last
CPU used by this session	456
CPU used when call started	456
index fast full scans (direct read)	0
index fast full scans (full)	0
index fast full scans (rowid ranges)	0
physical read bytes	270336
physical reads	33
physical reads cache	33
recursive calls	2558
recursive cpu usage	449
redo entries	4310
redo size	1315056
session logical reads	1145089
session pga memory	393216
session stored procedure space	0
session uga memory	196392
table fetch by rowid	4188
table scan rows gotten	38529196

Příloha E

Statistiky procedury zastřešující SQL řešení procházení síťového grafu

Name	Last
CPU used by this session	69
CPU used when call started	67
index fast full scans (direct read)	0
index fast full scans (full)	0
index fast full scans (rowid ranges)	0
physical read bytes	524288
physical reads	64
physical reads cache	64
recursive calls	289
recursive cpu usage	69
redo entries	745
redo size	249568
session logical reads	129547
session pga memory	131072
session stored procedure space	0
session uga memory	0
table fetch by rowid	1414
table scan rows gotten	4401797

Příloha F

Statistika procedury používající pohled pro průchod síťového grafu

Name	Last
CPU used by this session	37
CPU used when call started	36
index fast full scans (direct read)	0
index fast full scans (full)	0
index fast full scans (rowid ranges)	0
physical read bytes	16384
physical reads	2
physical reads cache	2
recursive calls	1967
recursive cpu usage	36
redo entries	2247
redo size	610900
session logical reads	17491
session pga memory	0
session stored procedure space	0
session uga memory	65464
table fetch by rowid	7389
table scan rows gotten	8

Příloha G

Porovnání časové náročnosti jednotlivých procedur pro práci se stromy

počet uzlů	čas [s]						
	rekurzivní funkce	dopředná funkce	rekurzivní SQL	strom s genealogickým id	generování genealogického id	nested sets	generování nested sets
100	0,016	0,078	0,015	1,031	2	0,76	0,9
1000	0,046	0,14	0,031	1,031	2,4	0,76	2,047
10000	0,5	0,85	0,14	1,031	3,5	0,76	16,97
100000	3,7	6,1	1,094	1,094	18,4	0,76	127,5
500000	43,6	66,4	44,9	1,4	112,1	0,76	773,2

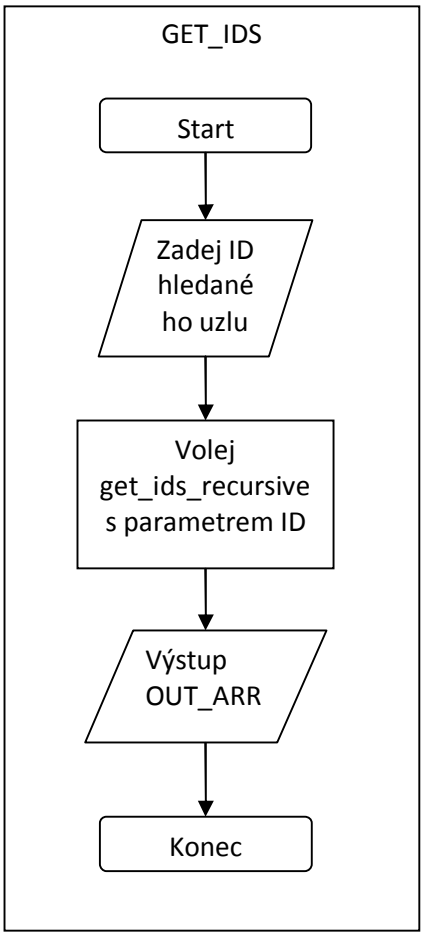
Příloha H

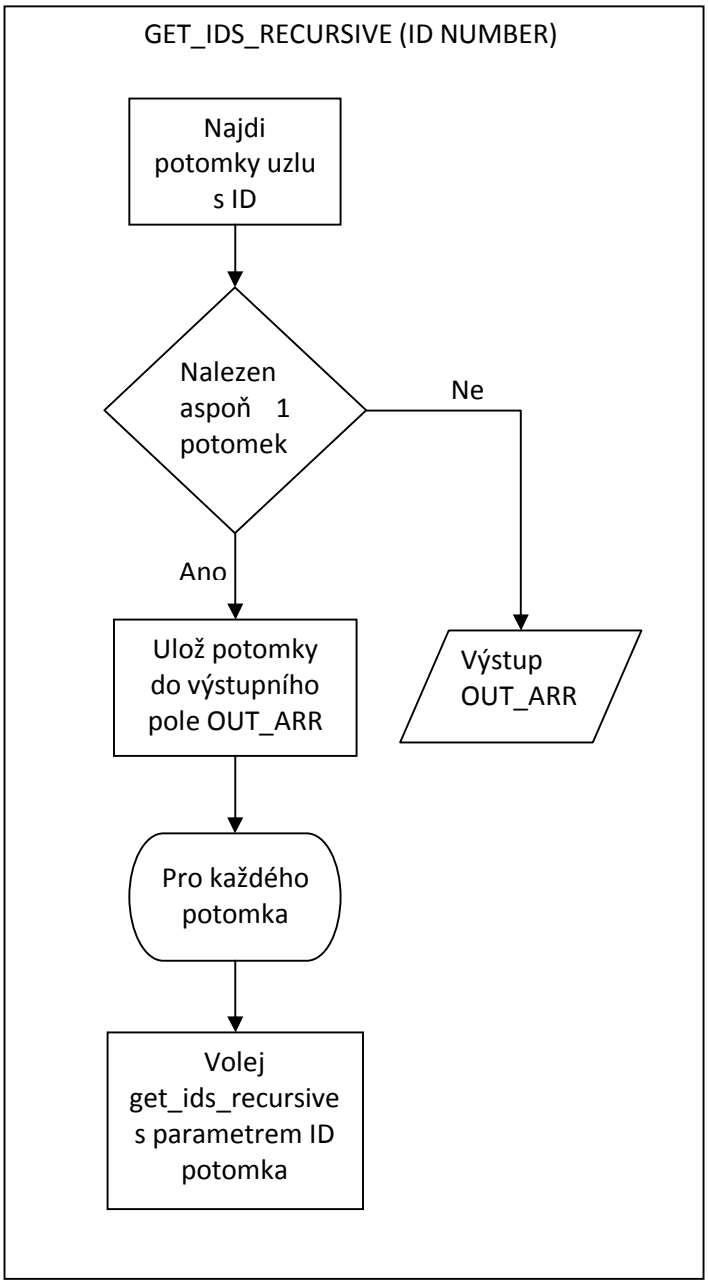
Porovnání jednotlivých metod zpracování síťových grafů

počet uzlů	počet vazeb	čas		
		Základní PL/SQL procedura	SQL řešení	S využitím pohledu
248	1163	2,427	1,302	0,39
457	2066	4,5	3,66	0,42
805	3739	9,2	6,36	0,656
5600	26857	>60	52	9

Příloha I

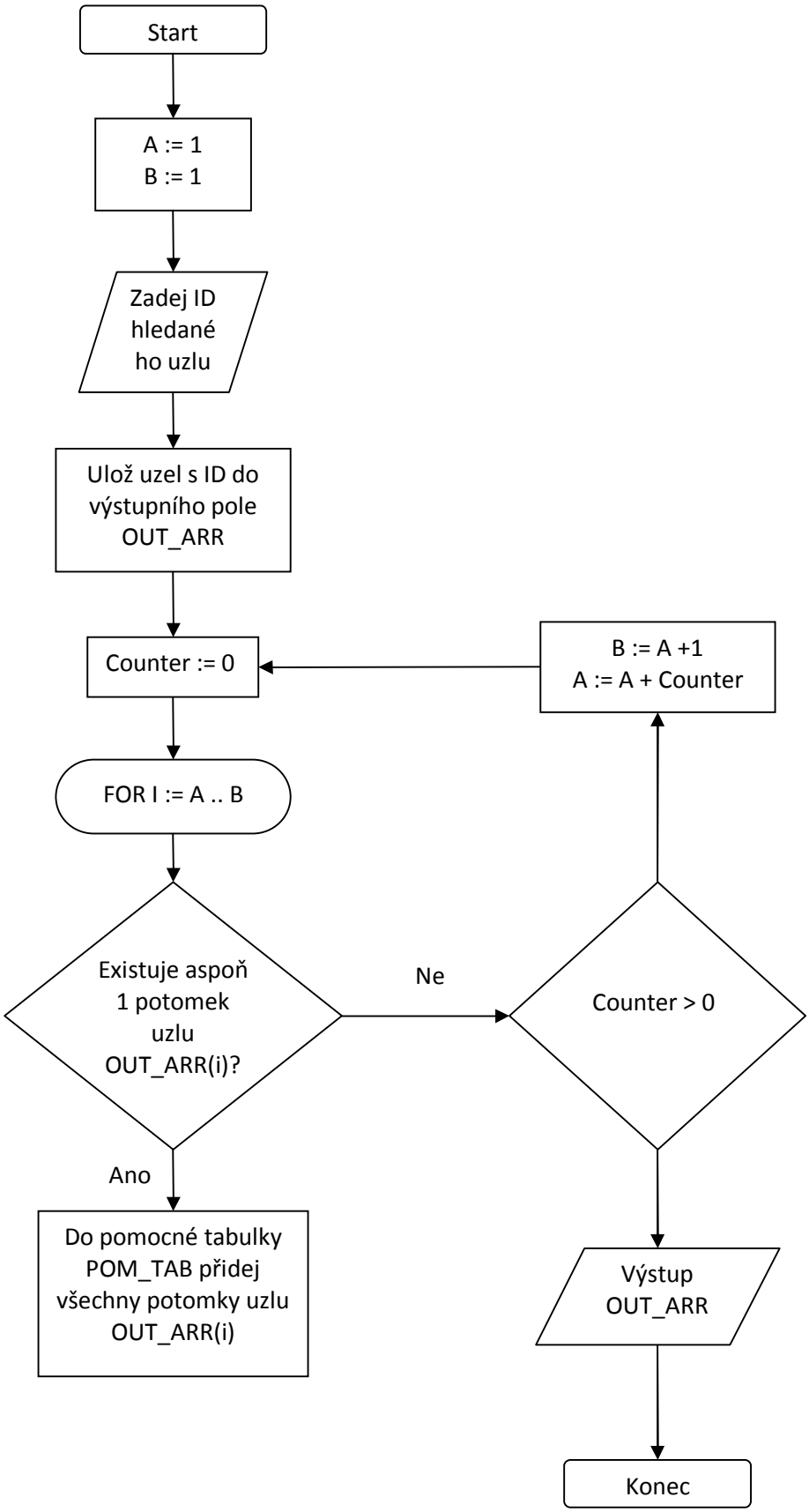
Vývojový diagram pro rekurzivní proceduru pro procházení stromu





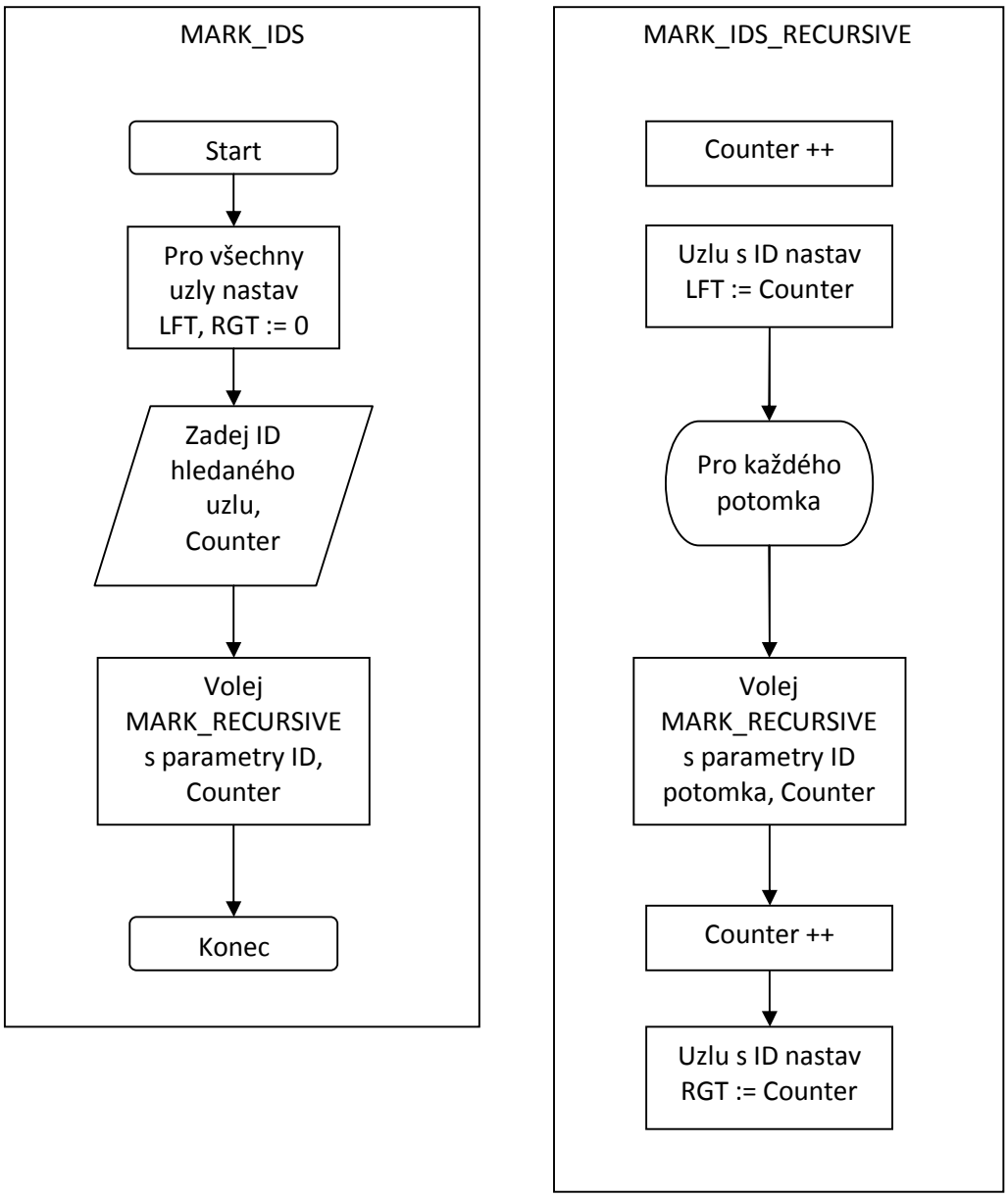
Příloha J

Vývojový diagram pro dopřednou proceduru pro procházení stromu



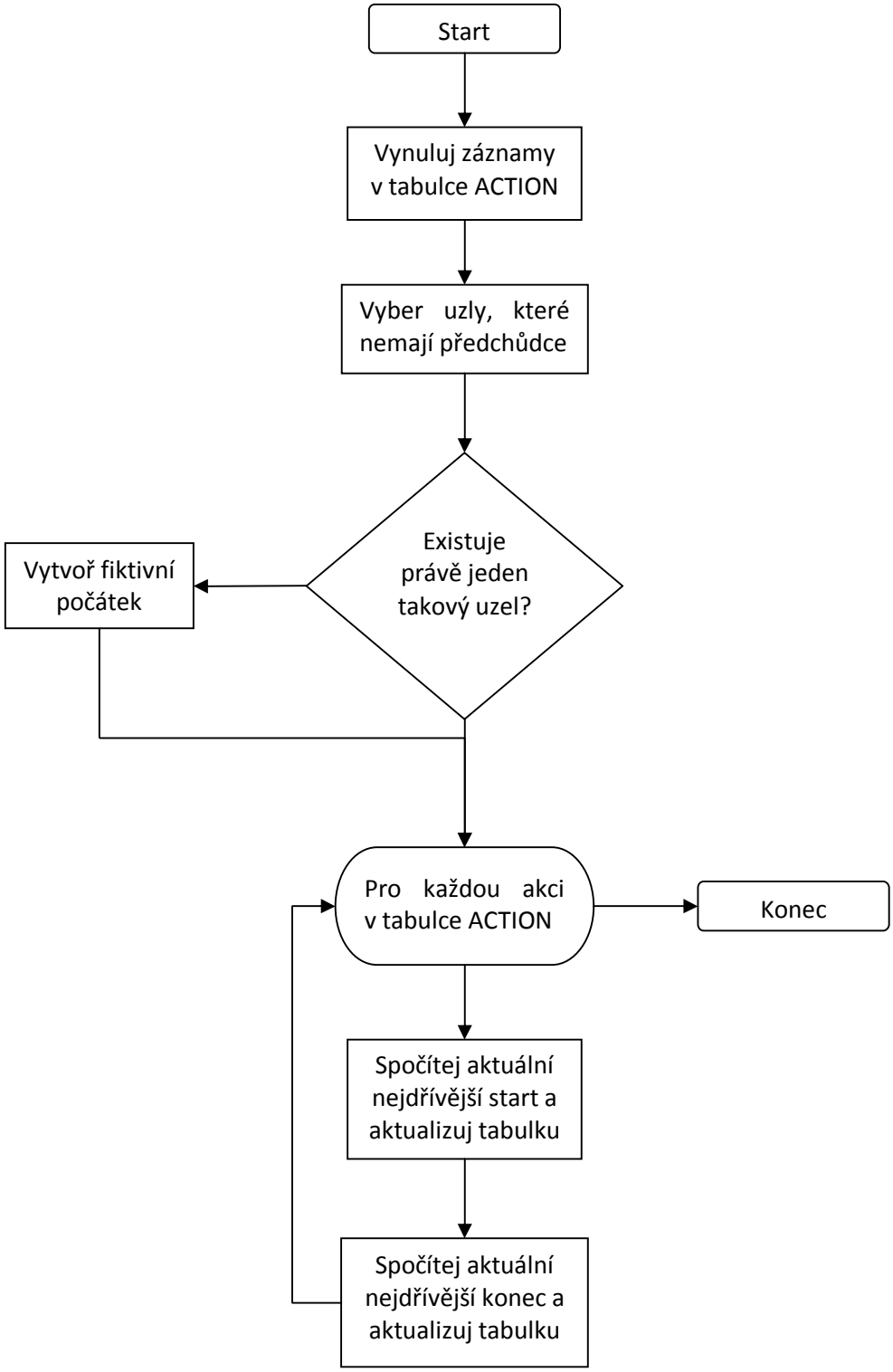
Příloha K

Rekurzivní procedura pro ohodnocení „Nested Sets“



Příloha L

Základní PL/SQL procedura pro procházení síťového grafu



Příloha M

Procedura pro procházení síťového grafu s využitím pohledu